BASIC TO ROM COMPILER MANUAL

Copyright (c) 1988
King Computer Services, Inc.

King Computer Services, Inc.
10350 Samoa Avenue, Tujunga, CA 91042
(818) 951-5240

RBASIC SUPPORT

PLEASE READ THIS AGREEMENT BEFORE COMMENCING USE OF RBASIC.

IF YOU DO NOT AGREE WITH THIS SUPPORT AGREEMENT, PLEASE RETURN YOUR RBASIC PACKAGE FOR A REFUND.

In our attempt to make the RBASIC package as inexpensive as possible, we have not included the provision for phone support in the cost of the package. However, since the Beta test was completed, we have found that all support calls have involved bugs in the BASIC code, not in the RBASIC Compiler itself. This is not to say that the Compiler is guaranteed bug-free, but a great deal of testing was done in Beta.

In view of the fact that it is extremely expensive, time consuming and sometimes impossible to sort out what the problem is with a program on the phone, we will not provide phone support for the RBASIC compiler.

We emphasize that before using RBASIC, you:

1. Thoroughly test your program.
2. Read the RBASIC Manual, making sure that you do not go past any words you do not understand. If you come across a word you do not understand, look it up in a good dictionary before proceeding.
3. Revise your code and make any necessary changes where RBASIC differs from Standard M100 BASIC.
4. Retest your program to ensure there are no bugs before using RBASIC.
5. After compiling your code with RBASIC, run it as a .CO program before trying to burn eproms.
6. Only when you are certain there are no problems with the code, burn your EPROM.

Proceeding in this fashion makes it easier to isolate just where the problem is.

If after you have thoroughly tested your program, you have a problem with the output using RBASIC, please do the following:

1. Write down exactly what the problem is.
2. Write down the line number where the the problem occurred.
3. Write down a brief description of what the code is supposed to be doing.
4. Send this with a copy of your program on a diskette to our Technical Support department.

We will endeavor to find out what the problem is and fix it. If it turns out to be a bug in RBASIC, we will return your program and the revised RBASIC compiler free of charge.

If the problem is not caused by RBASIC, but by some other problem, your program will be returned with notes on what we found. In common with the policy of many software houses, if the problem is not caused by a bug in our product we will bill for time spent on the problem. Our standard rate for this service is $60.00/hr, in 15 minute increments.

After an initial review, our technical support specialist will have some idea of how long the fix will take. If he estimates it will be lengthy to locate and fix the problem, we will contact you to find out how you wish to proceed.

King Computer Services, Inc.
1016 North New Hampshire, Los Angeles, CA 90029

Terms and conditions of sale and license of King Computer Services, Inc. (KCSI) Software purchased from King Computer Services or an authorized dealer.

LIMITED WARRANTY

I. Customer Obligations.

A. Customer assumes full responsibility that this software meets the specifications, capacity, capabilities, versatility and other requirements of customer.

B. Customer assumes responsibility for the conditions and effectiveness of the operating environment in which the software is to function, and for its installation.

II. KCSI Limited Warranty.

A. KCSI makes no warranty as to the design, capability, capacity or suitability for use of the software. The software is licensed on an AS IS basis, without warranty other than for the media.

B. No one is authorized to alter the terms of this warranty.

III. Limit of Liability.

A. KCSI shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be cause directly or indirectly by the software.

B. KCSI will not be liable for any damages caused by delay in delivering or supplying the software.

IV. RBASIC Compiler Software License.

A. The Copyright on the RBASIC Compiler is held by King Computer Services, Inc.

B. Applicable copyright laws apply to the software.

C. Title to the software remains with King Computer Services, Inc.

D. The customer may use the Software on one host computer.

E. The customer is permitted to make additional copies of the software only for backup purposes.

V. RBASIC Libraries Software License.

A. The Copyright for the Libraries is held by King Computer Services, Inc.

B. Applicable copyright laws apply to the software.

C. Title to the software remains with King Computer Services,Inc.

D. The software is intended to be used as an integral part of (linked with) software created by the customer. When used in such a format, it may be distributed without further licensing or royalty.

E. The Library files (.REL) files themselves may not be distributed, or reproduced in any way but may be copied only for back up purposes.

Table of Contents

Table of Contents

Table of Contents

Installation

Distribution Diskette.

The distribution diskette contains the following files.

1.	RBASIC.EXE	The BASIC to ROM compiler (translator).
2.	A100.EXE	An 8085 Assembler used to assemble the output from RBASIC.EXE.
3.	RCV.EXE	A PC program to upload BASIC files from the Model 100/102 to the PC.
4.	SND.EXE	A PC program to download .HEX files from the PC to the Model 100/102.
5.	COLOADER.EXE	A PC program that will create one of two loader programs RCO.BA or RRM.BA that can be sent to the Model 100/102.
6.	RCO.ASM	The source code for an assembly language program that is used by COLOADER.EXE to create RCO.BA.
7.	RRM.ASM	The source code for an assembly language program that is used by COLOADER.EXE to create RRM.BA
8.	xxxxxxxx.SUP	Several support files containing Assembly language source code support routines that are included in the assembly by the A100 assembler.

9.	xxxxxxxx.100	Support files that are specific to the Model 100/102.
10.	xxxxxxxx.CO,ROM	Support files that are specific to the Model 100/102 that require different versions depending on whether the file is being compiled as a .CO file or a ROM image.
11.	xxxxxxxx.BA	One or more BASIC files that may be used for testing the compiler.
12.	xxxxxxxx.DOC	Document files relating to the sample BASIC programs.
13.	README.RBS	This file if included will contain any changes to the information in this manual.

Hard Drive Installation.

Working with a hard drive is preferable but not essential. Simply create a directory on your hard drive to contain the files. The directory may have any name, but this example uses RBASIC.

1. Ensure you are currently logged in to the hard drive by typing

   C: <ENTER>

2. Change to the root directory by typing

   cd \ <ENTER>

3. Create the new directory by typing

   md rbasic <ENTER>

4. Change into the newly created directory by typing

   cd rbasic <ENTER>

5. Insert the distribution diskette in drive A:.

6. Copy all the data from drive A: to the new directory by typing

   copy A:*.* <ENTER>

7. Put the distribution diskette away and save it as your backup.

Dual Floppy Installation.

1. Insert a blank diskette in drive B:

2. Format the diskette by typing

   format b: <ENTER>

3. Press enter when the format program asks you to insert the diskette in drive B:.

4. When the format program is done it will ask you if you want to format another. Type 'N' and press enter.

5. Insert the distribution diskette in drive A:.

6. Copy all the files to drive B: by typing

   copy A:*.* B: <ENTER>

7. When the copy is completed, remove the distribution diskette from drive A: and save it as your back up.

8. Remove the diskette you have just created from drive B: and label it "RBASIC WORKING DISKETTE".

An Overview of RBASIC

About This Manual.

Some of the aspects of using RBASIC are very technical. While every effort has been made to keep it simple, you will be exposed to some assembly language coding concepts and some of the areas of the Model 100/102 Operating System that you do not normally have to deal with when developing interpreted BASIC code.

It is important therefore that you read this manual and fully understand the differences between RBASIC and BASIC and how the development cycle works before you plunge in to your code. Without prior assembly language programming experience, you can end up over your head very quickly if you do not read this manual.

Please read the manual.

Why use RBASIC.

The primary reason for using RBASIC is the ease with which it allows Model 100/102 programs to be moved into and run from option ROMs. It will allow very large programs to be compiled and loaded into an option ROM, thus freeing up RAM for data files.

RBASIC is a semi-compiler. It does not produce true compiled code. Large portions of code are left for the BASIC interpreter to handle. It may produce an improvement in execution speed because of the amount of code that is actually compiled. Certain key verbs such as GOTO, GOSUB and RETURN that normally have to be interpreted by BASIC are compiled to faster machine language equivalents.

Every statement (initial keyword in a command) is called directly. This means that the interpreter does not have to search through tables to determine which command to execute. In some areas, notably FOR-NEXT handling, RBASIC runs slower than interpreted BASIC, and this may offset the speed gained in other areas.

RBASIC creates code security. Whether you use RBASIC to generate ROMable code or .CO programs, the program cannot be LISTed. It is conceivable that someone could disassemble the program, but it is much harder to decipher what is happening in the program because of the way the code is generated.

RBASIC creates code integrity. When you have a critical application on the line such as a process control, the last thing you want is some bright spark delving into the code to improve performance by changing the program.

RBASIC is royalty free. Once you buy the package which is competitively priced, you may create, modify and burn as many ROMs as you want and you pay nothing further for the use of the program.

RBASIC is a no strings attached package. We do not attempt to nor do we pretend to own any portion of your code. You may use RBASIC as you would any other compiler. What you develop with it is yours to keep.

For the assembly language programmer, the RBASIC compiler produces commented assembly language listings that allow hand modification to handle special code requirements and the inclusion of machine language routines that may be assembled directly into the final output.

Operating Description.

Briefly the steps for the development of a BASIC ROM are.

1. Write a BASIC program on the Model 100 that follows the simple syntax guidelines for RBASIC. Run it and test it until it works to your satisfaction. Remember when testing that the final program will be committed to hardware (a ROM), and it is much harder to correct errors once the ROM is burned.

2. Upload the program to the PC.

3. Compile (translate) the program using RBASIC specifying that the output is to be used to create a .CO program. The output from the compiler will be an assembly language source code file ORGed in high memory that makes direct calls to Model 100/102 operating system routines.

4. Assemble the file created by RBASIC using A100.

5. Download the assembled program to the Model 100/102 and run it as a .CO program to ensure that the translation worked correctly. Test and correct as needed.

6. Compile (translate) the program using RBASIC specifying that the output is to be used to create a ROM image program. The output from the compiler will be an assembly language source code file ORGed at address 0 in memory that makes indirect calls to Model 100/102 operating system routines. The file will also include the necessary ROM support routines.

7. Assemble the file created by RBASIC using A100.

8. Download the assembled program to a ROM burner, or ROM Emulator such as the SoundSight Chameleon, or the GTEK ROM Emulator.
   If you have downloaded to a ROM emulator you may do further testing and correction before finally committing the result to a ROM.

The RBASIC package includes send and receive programs (SND.EXE and RCV.EXE) designed to send from and receive to a PC clone. They are included for convenience, but it is recommended that you use your own telecommunications program to do the transfers. SND.EXE and RCV.EXE do not work with all clones and if either of them doesn't work then you will have to provide a TC package of your own. Every package I have seen supports ASCII transfers (with XON/XOFF flow control) which is the type of transfer needed to work with RBASIC.

Using RBASIC

Developing.

Before you begin writing or modifying programs, study the chapter on syntax differences. Every effort has been made to allow all of the Model 100/102 BASIC syntax to be used in a program, but there are a handful of exceptions and rules that you need to know.

Using the guidelines, develop or modify your program to conform to these rules. Test the program thoroughly to ensure that it runs to your satisfaction. The RBASIC compiler cannot detect logic errors in a program, and it assumes that it is compiling a program that will run on a Model 100/102.

Uploading.

Once the program is running satisfactorily, connect the PC and Model 100/102 via their RS-232 ports using a null modem cable or a straight cable with an null modem adapter.

Change to the directory (for hard drive) or the drive (for dual floppy systems) that contains the RBASIC development system.

Assuming the program is named PROGA.BA, on the PC type

```
RCV PROGA.BA                <ENTER>
```

On the model 100 from the BASIC screen type

```
SAVE "COM:88N1E"            <ENTER>
```

The RCV.EXE program will report the progress of the upload, and will inform you when the upload is completed.

The BASIC program will now exist on the PC as an ASCII file named PROGA.BA

Refer to the Chapter on Receiving files for further information on using RCV.EXE or if you are having trouble receiving. Some PCs cannot receive comfortably at 9600 baud and it will be necessary to change the baud rate as covered in that chapter.

Compiling.

The first step is to compile the program as a .CO file. The RBASIC command line supports various switches which are covered in detail in the chapter on compiler options. For the time being we will concentrate on three.

The /O switch allows you to specify an ORG address other than the default of 0 (zero). Any time the ORG address is set to zero, the compiler assumes that it will generate a ROM file. Any address other than zero will generate a .CO type program. The output .CO file from the compiler is anywhere from 10% to 50% larger than the original BASIC program. For the time being calculate the size of your original BASIC program and add about 50% to the size. Subtract this value from 62960 (MAXRAM) and that will be the ORG address for the .CO file. Assuming PROGA.BA on the Model 100/102 is about 2000 bytes, we will reserve 3000 bytes for the program and set the ORG address at 59000 (a nice round number). The switch for this will be /O59000. See the detailed description of the /O switch in the chapter on switches for more information on ORG addresses.

If the program uses ON ERROR logic, the compiler needs to know this. ON ERROR logic generates extra code for each statement in order to allow error trapping. The ON ERROR switch is /E.

If the program uses interrupt trapping logic ON COM, ON TIME$ or ON KEY, the compiler also needs to know this to generate additional interrupt checking logic. The ON INTERRUPT switch is /I.

RBASIC assumes a default file extension of .BA so the command line to compile the program would be at least

```
RBASIC PROGA /O59000
```

If the other switches are needed, they must be separated by spaces.

```
RBASIC PROGA /O59000 /E /I
```

Assuming there are no errors, the output file will be called PROGA.ASM and will be an assembly language source code file ready for assembly. If you have errors, please refer to the chapter on errors which contains descriptions on how to handle them.

Assembling.

The A100 assembler is a very rudimentary assembler designed to handle the output from the RBASIC compiler it has no switches or options. It does support some special non-standard opcodes for compiling ROM code. Please refer to the chapter on the assembler for a more detailed description.

The PROGA.ASM file would be assembled using the command

```
A100 PROGA                <ENTER>
```

The output from an assembly is three files. PROGA.SYM contains the addresses and numeric values for all labels and EQUate operators in the .ASM source code file. PROGA.PRN contains a listing file that contains a complete hex map of the assembled program including all INCLUDE files. PROGA.HEX is an Intel HEX file suitable for downloading to the Model 100/102 for testing.

Downloading.

The PROGA.HEX file may now be downloaded to the Model 100/102 for testing. The Model 100/102 requires a loader that can take the Intel .HEX file and poke it in to memory as a machine language program. COLOADER.EXE has been provided to create the loader for you. The loader must be set up at an address that is 500 bytes less than the ORG address used for the program you will be testing. PROGA in the example above was ORGed at 59000 so on the PC type

```
COLOADER RCO /o58500
```

This will use RCO.ASM on the distribution diskette to create a BASIC program named RCO.LDR.

On the Model 100 type

```
RUN "COM:88n1e            <ENTER>
```

From the PC type

```
SND RCO.LDR               <ENTER>
```

Once the program is loaded it will run on the Model 100 and display the message "Creating RCO.CO Program" and will return to the BASIC 'Ok' prompt when it is completed. You may delete the RCO.BA program by typing NEW. Your Menu will now show a new file RCO.CO. This is a machine language loader that will quickly download the main program for you.

Now we are ready to send the main program. position the cursor over RCO.CO and press ENTER. The screen will go blank and the Model 100/102 is now in receive mode waiting for you to send it a file. From the PC type

```
SND PROGA.HEX              <ENTER>
```

RCO.CO will display dots '.' on the screen as records are received from the PC. When the download is complete, you will be returned to the Model 100/102 menu.

This completes the downloading process. For further information on SND.EXE, consult the chapter on sending files from the PC and on receiving on the Model 100/102.

Testing.

Once the program is downloaded it can be run by calling the entry address. For this particular example on the Model 100/102 type

```
CALL 59000                 <ENTER>
```

The program should run exactly as it did when it was a BASIC program. Test all the ins and outs of the program. Error trapping logic and any Interrupt logic. Ensure that free space has not been used up when exiting the program.

If there are any problems with the program, try to isolate the line or lines causing the problem. See the Trace switch and Line number switch in the Compiler switches chapter and try re-compiling and running with trace on to isolate the problem. Check the syntax of the line(s) in question and follow the guides in the chapter on errors. Once the program is running correctly the final compile can be done.

Compiling for ROM.

Compiling for the ROM version is identical to compiling for a .CO version, except that the ORG address is omitted from the command line.

```
RBASIC PROGA /E /I        <ENTER>
```

There is one more thing that you may want to do for the ROM version. Once a program is burned in to a ROM, it is started by CALLing 63012. Popular commercial ROMs shortcut this approach by having the ROM install a name for itself on the Model 100/102 menu which may be used to start the ROM. The ROM compiler allows this option using the /R switch followed by up to 6 characters of name that will be placed on the menu exactly as you enter them.

```
RBASIC PROGA /E /I /RProga         <ENTER>
```

The above line will compile the program and add in code to create a menu entry of "Proga" on the Model 100/102 menu that may be used to start the ROM.

When the ROM is first plugged in, it is started by

```
CALL 63012                <ENTER>
```

Once this is done, the name "Proga" will appear on the menu can be used to start the ROM.

The output file from RBASIC will be called PROGA.ASM and will be larger than the original PROGA file because of the addition of special code to support the Option ROM logic.

PROGA.ASM is assembled with A100

```
A100 PROGA                <ENTER>
```

which will again create the files PROGA.SYM, PROGA.PRN and PROGA.HEX.

PROGA.HEX is in Intel Hex format, a very standard format used by virtually every ROM burner on the market. This file can be downloaded to the burner in whatever way is usual for that burner and the ROM can be created.

RBASIC Syntax

General Syntax Rules.

In general, RBASIC supports all of the 100/102 syntax that would apply to a compiled BASIC. It does not support commands that relate only to interpreted BASIC such as MERGE, LIST and LLIST. It also does not support the undocumented syntax of 100/102 BASIC such as NEXT C,B,A.

There are a few code ordering requirements but in general you will not find much difference between BASIC and RBASIC because a lot of work went into keeping them compatible to allow easy program conversion.

Unsupported Commands.

The only commands that are not supported in RBASIC are the ones that clearly relate to interpreted BASIC. These are SAVE, CSAVE, LIST, LLIST, MERGE, EDIT, CONT, DSKO$ and NEW.

The IPL command is not supported, but is intended to be included in a future release.

The MERGE verb is used for including assembly language source code files in an RBASIC program. Please refer to the detailed chapter on using the MERGE verb.

Commands Requiring Modified Syntax.

FOR-NEXT.

FOR-NEXT loops within the code are actually rewritten and compiled into a loop with a control variable as follows

Original Code:

```
10 FOR A = 1 TO 10
20 'Do Something
30 NEXT A
40 FOR B = 100 TO 1 STEP -1
50 'Do Something
60 NEXT B
70 FOR A = C TO D STEP E
80 'Do Something
90 NEXT A
```

Becomes:

```
10 LET A = 1
20 'Do Something
30 LET A=A+1:IF NOT (A>10) THEN GOTO 20
40 LET B=100
50 'Do Something
60 LET B=B+(-1):IF NOT (B<1) THEN GOTO 50
70 LET A=C
80 'Do Something
90 LET A=A+E:IF NOT (A>D) THEN GOTO 80
```

The above is representative of the way a FOR-NEXT will behave though the actual technique used produces code that is a little tighter than the above example.

The previous examples all show FOR-NEXT loops starting on a new line number for illustration only. FOR-NEXT controls may be placed anywhere on the line as in

```
10 PRINT "START":FOR A = 1 TO 10:PRINT A;:NEXT A
```

Using this construction imposes some limitations on syntax, variable use and timing.

First any variables used in a FOR-NEXT control should not be modified within the loop.

```
10 B=100
20 FOR A=1 TO B
30 B=B+1
40 NEXT A
```

The above code in BASIC will correctly cause A to be incremented to 100 and the loop will end. BASIC evaluates the value of B once at line 20 and then ignores all changes to B in line 30.

In RBASIC the above will produce an infinite loop. Since A is compared to B each time a NEXT is issued, A will never become greater than B. If a step variable is used, it is subject to the same restrictions.

```
10 E = 1
20 FOR A=1 TO 10 STEP E
30 E=E+1
40 NEXT A
```

The above causes no problem in interpreted BASIC, but in RBASIC the step value will be increased by 1 on each pass. The loop will execute less than 10 times because of the change in the step variable.

Timing problems will arise when the limit or the step values are variables rather than constants.

```
20 FOR A = B TO C STEP D
30 'Do Something
40 NEXT A
```

Each time the NEXT is executed, the Variables A, C and D must be looked up in the variable table. The amount of time the look up will take depends on where the variables are located in the table. A variable is added to the table the first time that it is referenced, so the timing on the FOR-NEXT could behave differently at different times in the program if the variables do not already exist.

If timing is critical, force the variables into existence near the beginning of the program logic so that their locations are fixed early in the variable table before they are used anywhere else in the program. Once this is done, the timing of the loop will be consistent. You will have to time the loop to figure out what the value will be, but the value will not change as long as the variables are initialized at the top of the program.

```
10 A=0:B=1:C=10:D=1
20 FOR A = B TO C STEP D
30 'Do Something
40 NEXT A
```

Syntax differences.

1. Negative Step values are recognized only when the compiler encounters a '-' as the first non-space character after a STEP statement. The '-' must be explicitly placed in the source code. If a STEP is to be down rather than up, then the first character after a STEP must be a minus '-'. If the STEP value is a negative variable then it must be made positive and preceded with a '-'.

```
BASIC
10 E = -15
20 FOR A = 100 TO 1 STEP E
30 'Do Something
40 NEXT A
```

```
RBASIC
10 E = -15
15 E = -E
20 FOR A = 100 TO 1 STEP -E
30 'Do Something
40 NEXT A
        or
10 E = -15
20 FOR A = 100 TO 1 STEP -(-E)
30 'Do Something
40 NEXT A
```

Note that in both RBASIC examples the first non-blank character after the STEP keyword is a minus '-'. The compiler uses this character to recognize a downward stepping FOR-NEXT.

Other Syntax differences in FOR-NEXT.

2. The NEXT in a FOR-NEXT control statement must always name the control variable.

```
BASIC:
10 FOR A = 1 to 10: GOSUB 100 : NEXT
```

```
RBASIC:
10 FOR A = 1 to 10: GOSUB 100 : NEXT A
```

The absence of the variable name will result in a Missing or Invalid NEXT variable Error during compilation.

3. The NEXT in a FOR-NEXT control statement must always appear after the FOR statement. BASIC allows GOTOs to be used to force unusual constructions which RBASIC will not support.

BASIC:
```
10 GOTO 30
20 NEXT : GOTO 40
30 FOR A = 1 to 10: GOSUB 100: GOTO 20
40 'Do Something Else
```

RBASIC:
```
10 FOR A = 1 to 10
20 GOSUB 100: NEXT A
30 'Do Something Else
```

4. Whenever RBASIC encounters a NEXT, it operates strictly on the assumption that the innermost currently active FOR is to be taken to its NEXT state.

The following will work in BASIC but not in RBASIC.

```
10 FOR A = 1 TO 10
20 'Do Something
30 IF (X<>24) THEN NEXT A ELSE NEXT A:PRINT "X=24"
```

The intention of the above code is to complete the FOR-NEXT loop and carry on to the next line if X=24 or to complete and print a message if X<>24.

RBASIC cannot handle this syntax. It sees the first NEXT A as the complement to line 10. The second NEXT A in line 30 produces a NEXT with FOR error. This logic must be rewritten to work in RBASIC.

```
10 FOR A = 1 TO 10
20 'Do Something
30 NEXT:IF X=24 THEN PRINT "X=24"
```

DATA

1. DATA statements must appear on their own lines and not mixed with other statements.

BASIC:

```
10 PRINT "HELLO":DATA 1,2,3
```

RBASIC

```
10 PRINT "HELLO"
20 DATA 1,2,3
```

2. BASIC allows strings not enclosed in quotes as data statements. RBASIC requires that all strings be quoted.

BASIC:

```
10 DATA HELLO,BYE,10
```

RBASIC:

```
10 DATA "HELLO","BYE",10
```

3. RBASIC will not recognize an OUT OF DATA condition. During compilation, RBASIC collects up all data statements and moves them to the end of the BASIC code. When a READ command is issued the next piece of stored DATA is read in exactly as is done in interpreted BASIC. If a READ goes beyond the end of the DATA statements, the program just keeps trying to read in whatever else is in memory which is usually support code. It is up to the programmer to ensure that his data statements are correctly matched with read statements.

The moving of DATA statements in an RBASIC program does not affect the way your program behaves.

CALL

One of the undocumented features of the CALL Statement that is used by many programmers is the fact that on entry to the CALL, the DE register pair points to the remainder of the CALL line. Several programs use this to pass a third parameter, using the documented A and HL register pairs as the first and second parameter after a CALL and then placing a commented string at the end of the line. DE will point to the string.

```
CALL 55078,5,22 'george
```

On entry at 55078, the A register contains 5, the HL register contains 22 and the DE register points to "'george".

Generally in RBASIC comments are removed during the compilation.

Because the above technique is commonly in use, an exception is made for comments following a CALL statement. These are included in such a way that the DE register ends up correctly pointed at the comments.

If you do not use the technique, remember that comments after a CALL are included in the output program and will take up space.

Commands That are Treated Differently.

END, and STOP are both treated as an END statement.

The MENU command will cause the program to stop running and return to the Model 100/102 main menu.

CLEAR

RBASIC supports all three versions of the CLEAR command but in different ways. It is important to understand the differences.

1. CLEAR may be used in RBASIC without arguments to zero out all variables.

```
10 CLEAR
```

This causes all numeric and string variables to be removed from the variable table. As far as the program is concerned, this action is the same as setting all strings to a value of "" and all numbers to a value of 0 (zero). In addition all arrays are unDIMed by this verb.

This is the only version of CLEAR that has this effect. In interpreted BASIC any version of CLEAR causes variables to be erased. In RBASIC the CLEAR verb must be used on it's own to erase variables.

Another difference is that CLEAR used anywhere in interpreted BASIC does more than initialize variables, causing the stack to be destroyed including all pending RETURNs and FOR-NEXT loops. The RBASIC CLEAR only clears variables and DIMs.

2. CLEAR followed by an integer is used to allocate string variable space.

```
CLEAR 500
```

may be used to clear string space. Please see later notes on clearing extra string space for RBASIC programs.

The CLEAR string space command is compiled so that it only occurs once in the running program. If a program contains

```
10 CLEAR 100
        and
40 CLEAR 500
```

RBASIC will compile this to a single CLEAR 500 to allocate string space as part of the set up actions by the compiled program before the first line of code is executed. If no CLEAR command is issued, the compiler compiles a CLEAR 256 into the code.

As mentioned CLEARing string space is not executed as part of the running program but as part of the set up logic, and the CLEAR nnn does not cause variables to be reset.

In interpreted BASIC

```
10 CLEAR 200               'Clears all, allow 200 str bytes
```

In RBASIC

```
10 CLEAR:CLEAR 200         'Clear all, allow 200 str bytes
```

In practice most CLEARing is done at the start of a program before any variables have been initialized, and there is no need to repeat the CLEAR verb.

3. As of version 2.00, clearing HIMEM is supported by RBASIC. The syntax is the same as interpreted BASIC

```
10 CLEAR 500,55000
        or
20 CLEAR ,52000
```

The first line will set up 500 bytes of string space and set HIMEM to 55000. The second version sets up HIMEM without allocating the string space.

All the CLEAR values for HIMEM in a program are processed and the lowest memory location is used to set HIMEM once before the program begins running. If lines 10 and 20 above both appeared in a program, HIMEM would be set only once to 52000, the lower value, before the program began running.

An RBASIC program compiled as a .CO file will itself attempt to occupy some sort of HIMEM position. If the CLEAR statement attempts to set HIMEM above the beginning of the RBASIC program, the statement will be ignored and the compiler will issue a warning.

An example will illustrate this better than an explanation. Assume that PROGA.BA contained line 20 above to set HIMEM to 52000. If the PROGA.BA itself is compiled at an address below 52000 such as

```
RBASIC PROGA /o49000
```

it would be dangerous for the program to reset HIMEM to 52000. In a case like this, the CLEAR value is not compiled and the compiler provides a warning message that an attempt was made to CLEAR a HIMEM value higher than the ORG address of the program.

This type of problem will usually occur when the final RBASIC program will reside in a ROM and will need to CLEAR some HIMEM, but the test version is assembled as a .CO file.

Once the program is compiled and assembled as a ROM program, the warning will disappear and the CLEAR statement will compile correctly.

CLEAR will not accept variable arguments for either string space or HIMEM.

Interpreted BASIC allows

```
10 A=200:C=55000
20 CLEAR A,C
```

RBASIC requires that either arguments when used must be constants.

MAXFILES

MAXFILES may only be issued once in the program and behaves like the CLEAR nnnnn command. The largest of multiple MAXFILES statements is used, and is processed in the set up code before the first line of the program is executed. If no MAXFILES command is issued the compiler generates code for MAXFILES=1. MAXFILES like CLEAR will not accept a variable as the argument, but only a constant in the range 0 to 15.

GOSUB, GOTO

Because BASIC is interpreted, it will let you get away with missing lines such as

```
10 GOTO 30
20 GOSUB 100
30 PRINT "HELLO"
40 END
```

Because line 20 is never executed, the interpreter does not have to deal with the fact that there is no line 100 for the GOSUB. RBASIC must resolve all line numbers into addresses at compile time. Each line number is given a label of its original line number preceded by an 'L'. Line 20 above will generate code that reads

```
L20:
        CALL     L100
```

The A100 assembler will have a problem with this because L100 is never defined later in the program. The compiler will compile the program correctly, but the assembler will issue an error of

```
Undefined Label L100
```

When you see this error, your original program will contain a GOSUB or GOTO to a line that does not exist.

Handling String Space.

This is a primary area of difference between RBASIC and BASIC.

In BASIC, an area of string space is cleared in memory and certain string variables, but not all, are placed in this area.

```
10 CLEAR 100
20 B$=STRING$(100,'*')
30 A$="ABCDEFGHIJKLMNOP"
```

The above commands will work in BASIC even though it appears that too much string space is being used. Line 20 uses up the 100 bytes of string space, but line 30 does not require string space. The variable A$ created in line 30 is set up to point to the actual "ABCDEFGHIJKLMNOP" string in the BASIC program. If a string requires a computation or an operation to create it or to determine its value, it is allocated to string space. If it is a straight assignment - a constant, string space is not allocated, and the string in the BASIC code is used as the variable. READ-DATA statements are treated the same way.

```
10 CLEAR 100
20 READ A$
30 END
40 DATA "HELLO YOU ALL"
```

The READ at line 20 does not use any string variable space, because A$ is set to point straight to the string in the DATA statement in line 40.

RBASIC is different because it requires all strings to be created in the memory reserved for string space. Once RBASIC code resides in the Option ROM the Standard ROM cannot point to or access memory in the option ROM.

The compiler handles this by recognizing string constant assignments and converting them to string operations.

```
10 LET A$= "HELLO"
        becomes
10 LET A$="HELLO"+""      ("HELLO" plus a null string)
```

The compiler recognizes the above as a constant and will automatically generate the above code causing all the string to end up in string variable space. This means that your program will probably require more string space.

The RBASIC compiler will attempt to count up string constants used in LET assignments or DATA statements and will display the information at the end of the compilation as:

String Constants = nnn bytes

The nnn value will be a decimal number composed of the bytes counted that appear as constants in LET and DATA statements.

```
10 A$="Hello"
20 PRINT A$
30 A$=A$+"You all"
40 PRINT A$
50 READ A$
60 PRINT A$
70 END
80 DATA "Goodbye"
```

After compiling the above program, RBASIC will display the message:

String Constants = 12 bytes

This figure is derived by adding the bytes used by "Hello" and "Goodbye" in lines 10 and 80. The string "You all" in line 30 is not counted because it is not a constant.

In general terms, the byte count for string constants should be added to whatever value you were using as an argument to CLEAR in your original BASIC program. In the above program, if the original program CLEARed 100 bytes, this should be changed to CLEAR 112.

Remember that if you do not issue an explicit CLEAR command in an interpreted BASIC program, a default of 256 is used just as if you had included a command to CLEAR 256.

Undocumented Syntax.

The compiler was written around the documented BASIC syntax. There are a great number of undocumented BASIC syntax formats that the Model 100 will accept. The task of tracking through the Model 100/102 to find all the undocumented syntax is way beyond the scope of this compiler though some effort has been made to achieve compatibility.

For example the BASIC interpreter will accept the NEXT syntax in line 30 which is not mentioned in the manual.

```
10 FOR C = 1 to 10: FOR B = 1 to 10: FOR A = 1 to 10
20 STATEMENT: STATEMENT
30 NEXT A, B, C
```

The RBASIC compiler will support this NEXT usage, but the rule is if the syntax doesn't exist in the Model 100 BASIC manual then don't expect RBASIC to handle it.

The RBASIC compiler will accept an empty THEN statement. The following produces the same result in both the BASIC interpreter and the RBASIC compiler.

```
10 IF A = 1 THEN ELSE GOTO 30
20 STATEMENT
```

Comments and Copyrights

It is fairly common to embed copyrights and version numbers in a BASIC program using comments. Because RBASIC disposes of all comments, these would be lost in the normal course of compiling.

If you want to embed that information into your program or onto a ROM so that the information would show up if anyone disassembled the package, then you must do so by using an assignment statement that will will be compiled into the final output.

```
BASIC
10 ' The Singer Program
20 '    by Singer Associates
30 ' v 1.00 Copyright (c) 1989
```

```
RBASIC
10 A$="The Singer Program"
20 A$="    by Singer Associates"
30 A$=" v 1.00 Copyright (c) 1989"
```

The above code causes A$ to be repetively assigned different values, but the strings will be compiled into the output .CO or ROM program and will be visible if the program is disassembled. They cause a minor slow up at the start of the program, but they do allow you to put your 'brand' on your ROM products.

Compiler Switches

What is a Switch

The RBASIC compiler supports a number of options that can be controlled at compile time by placing switches on the command line. A switch is a group of characters separated from other elements of the command line by one or more spaces. A switch starts with the '/' character. This character is followed by a single character indicating which switch is to be controlled. The character is case insensitive. /E and /e are the same switch. A switch may be followed by one or more additional characters indicating further actions to take on the switch. A switch may appear any where on the command line as long as it is separated from other elements of the command line by spaces.

```
RBASIC /i PROGA /E

RBASIC /e /i PROGA

RBASIC PROGA   /I                /e
```

All of the above examples have the same effect as switches to the compiler.

Including Comments /C

The Comment switch will cause each line of BASIC code to be included as a comment in the output .ASM source file. This is useful for studying how the compiler generates its output. The default is not to include comments.

```
RBASIC PROGA /c
```

Noisy Compile /N

The Noisy switch will display each line of BASIC code on the screen as it is compiled.

```
RBASIC /N
```

Line Numbering /L

The compiler normally generates a single dummy line number of 9999. If a syntax error occurs anywhere in the program, the display will read "SN? in 9999" because all lines have been given this dummy number. The line number switch will cause the correct line number to be set up before each line is executed. Since this generates more code, it may not be needed except in the early stages of development to provide correct line numbers in case the program does drop into an error message. The Line switch is automatically turned on by either the Trace Switch or the Error Trapping switch.

```
RBASIC PROGA /L
```

Tracing Program Execution. /T

One of the features of BASIC interpreters on larger computers are TRON (trace on) and TROFF (trace off) commands to trace program execution in order to locate logic bugs.

A trace option has been provided for the RBASIC compiler. Compiling with the trace switch will cause additional code to be generated to display the currently executing line number on the LCD. The numbers appear surrounded by "<>" parentheses.

```
<10> <20> <100> <110> <120> etc.
```

Using the trace switch also turns on the line numbering switch as if you had used /L on the command line.

```
RBASIC PROGA /t
```

Slow Trace /S

The slow trace acts exactly like Trace except that it waits for you to press a key after displaying the line number, but before executing the line. See notes in Chapter 14 for bugs using slow trace with programs containing INKEY$ logic.

```
RBASIC PROGA /S
```

Error Trapping. /E

ON ERROR logic causes the need for additional code to save the current statement and next statement which are needed to process a RESUME or RESUME NEXT command. The compiler has no way of knowing that such code is needed until it hits an ON ERROR statement. If you have used ON ERROR logic in your program, then compile with the Error trapping option. The Error trapping option turns on Line numbering as if you had included /L on the command line. For further information see the chapter on Error trapping.

```
RBASIC PROGA /E
```

Interrupt Trapping /I

The compiler also needs to be alerted if ON COM, ON TIME$ or ON KEY interrupt trapping is being used in the program. This also generates additional code, to check for and handle the interrupt events. For further information on this subject see the chapter on Interrupt Trapping. The /I switch automatically enables the break (/B) switch.

```
RBASIC PROGA /I
```

Break-Pause Check /B

The default code generated by the compiler ignores the BREAK and PAUSE keys. This switch will cause code to be generated to do a Break-Pause check before each line of code similar to the technique used by the BASIC interpreter. The Break-Pause check logic is needed if your code includes ON KEY interrupt logic, so using the /I switch will also cause the inclusion of Break-Pause checking code.

```
RBASIC PROGA /b
```

Setting the ORG Address /Onnnnn

The default ORG address for the .ASM file is 0 (zero). When the ORG address is set to 0, the compiler assumes that a ROM is being generated and does three things. First it includes support code needed for ROM programs. Secondly it changes the way Standard ROM routines are called to call them indirectly. Third it changes the code so that strings that must be processed by the Standard ROM are copied out of the Option ROM to RAM and sets HL to point to them in RAM. When the ORG address is set to a non-zero value, the file is generated as though it were going to be loaded and run from RAM. Obviously the ORG Address to do any good would have to be set to a value in high RAM. If no ORG address switch is set, the file is ORGed at zero. The ORG address must be specified as a decimal value.

```
RBASIC PROGA /O59000
```

An ORG address is an ORiGin. This address is the lowest address in memory at which code for a program will be placed. All programs are placed in memory in order to be run. BASIC programs do not have to be a fixed addesses but .CO (machine language) programs must be. Since RBASIC converts a BASIC program to a machine language program, an origin address must be selected for the program. For an Option ROM version, the Origin address is always 0 (zero). For a RAM version (.CO), an address in high memory must be selected. The operating system uses addresses from 62960 on up, so any machine language program will have to be placed below this address. The A100 assembler will tell you how big the resulting machine program is and what the highest possible ORG address that could be used is.

Naming The ROM Chip. /Rssssss

A program in a ROM may be started by CALLing 63012 and this may be used whenever you want to start the program. It is also possible to create a ROM trigger file. A ROM trigger file is a file created on the 100/102 menu that can be used to start the ROM program. This technique is used in commercial ROMs wherein you will see the name "GldCrd", or "Forms" on the menu. Positioning the cursor over the name and pressing enter will start the ROM program. The same may be done with an RBASIC ROM. The ROM Trigger name should be no more than 6 characters long and if you include more than 6 the remainder will be ignored. The ROM Trigger Name will be created exactly as entered.

1. RBASIC PROGA /RProg

2. RBASIC PROGA /RSTUFF

   The first example will create a menu entry "Prog" and the second will create "STUFF".

   You may include a space in a ROM Name, but if you do, the entire switch must be enclosed in quotes.

3. RBASIC PROGA "/RMy Prg"

   The last example will create a ROM Name of "My Prg" on the menu.

Assembling With MAC /M

A large part of this project was developed on a PC running a CPM emulator. The Assembler used was the Digital Research Assembler MAC until A100 was completed. MAC supports macros and other useful tools that can be used for debugging. If you have an emulator and MAC, you may specify that you want the output .ASM file to be MAC compatible. This would be useful if you intend to hand modify any of the Assembler code, and would prefer to write with macros. This switch would also work for any assembler that is fully MAC compatible.

```
RBASIC PROGA /m
```

See the section on the A100 Assembler for a list of differences between MAC and A100.

Sending From The PC

The SND.EXE Program.

The SND.EXE program is designed to be a quick telecommunications program for sending ASCII files from the PC to another computer. You may of course use any Telcom program to do this and may have to if SND.EXE does not work with your PC, but SND.EXE is very convenient because most programs require load time and then ask you for communications parameters and finally let you name the file to send. SND.EXE is as simple as

```
SND PROGA.HEX              <ENTER>
```

The Model 100/102 or other receiving computer must be ready in receive mode before the SND program is invoked and the two computers must be connected via a null modem cable or connector.

To send a BASIC program from the PC to the Model 100, the BASIC program must exist on the PC in ASCII format (untokenized). On the Model 100/102 type

```
LOAD "COM:88N1E"           <ENTER>
```

On the PC type

```
SND PROGA.BA
```

The PC will display the BASIC program as it is sent including the ending CR-LF as control characters ^M^J.

SND.EXE Defaults.

SND.EXE assumes the natural defaults for sending an ASCII file. XON/XOFF is enabled and this parameter is not optional. Data is sent out COMM1 on the PC but this may be changed with switches. Baud is 9600, Data is 8 bits, Parity is set to none and 1 stop bit is used.

Command line switches may be used to change the defaults other than XON/XOFF protocol which is always enabled.

SND.EXE Switches

SND.EXE expects the name of the file to send, and there are three switches available.

The Comm Port switch allows you to change to COMM2. The switch is /C2. The 'C' may be upper or lower case and may appear anywhere on the command line.

SND PROGA.BA /c2

The data switch allows you to change baud rate, data bits, stop bits and parity.

The switch is a slash followed numbers and letters similar to the Model 100 COM string, /bdps where:

b = Baud Rate	8=9600, 7=4800, 6=2400, 5=1200, 4=600, 3=300
d = Data Bits	8 or 7
p = Parity	N=No Parity, E=Even Parity, O=Odd Parity
s = Stop Bits	1 or 2

Example for 300 baud, 7 Data, Even Parity, 1 Stop use:

SND HELLO.BA /37E1

The matching parameter on the Model 100 would be

LOAD "COM:37E1E"

Note that the Model 100 uses the same string with an 'E' added on to the end to Enable XON/XOFF. The 'E' is not required for SND.EXE because it always uses XON/XOFF protocol.

The third SND.EXE switch displays a brief description of the command syntax and switches.

SND /h

SND.EXE will also display help if there is an error on the command line or switches or the specified file cannot be found.

Receiving To The PC

The RCV.EXE Program.

The RCV.EXE program is designed to be a quick telecommunications program for receiving ASCII files on the PC from another computer. You may of course use any Telcom program to do this and may have to if RCV.EXE will not work with your PC, but RCV.EXE is very convenient because most programs require load time and then ask you for communications parameters and finally let you name the file to receive. RCV.EXE is as simple as

```
RCV PROGA.BA                <ENTER>
```

RCV.EXE must be run and ready before the Model 100/102 or other sending computer begins sending and the two computers must be connected via a null modem cable or connector.

To receive a BASIC program from the 100 on the PC type

```
RCV PROGA.BA
```

On the Model 100/102 type

```
SAVE "COM:88N1E"            <ENTER>
```

When the upload is complete, the BASIC program will exist on the PC in ASCII format (untokenized).

RCV.EXE Defaults.

RCV.EXE assumes the natural defaults for receiving an ASCII file. XON/XOFF is enabled and this parameter is not optional. Data is received at COMM1 on the PC but this may be changed with switches. Baud is 9600, Data is 8 bits, Parity is set to none and 1 stop bit is used.

Command line switches may be used to change the defaults other than XON/XOFF protocol which is always enabled.

RCV.EXE Switches

RCV.EXE expects the name of the file to receive, and there are three switches available.

The Comm Port switch allows you to change to COMM2. The switch is /C2. The 'C' may be upper or lower case and may appear anywhere on the command line.

```
RCV PROGA.BA /c2
```

The data switch allows you to change baud rate, data bits, stop bits and parity.

The switch is a slash followed numbers and letters similar to the Model 100 COM string, /bdps where:

b = Baud Rate	8=9600, 7=4800, 6=2400, 5=1200, 4=600, 3=300
d = Data Bits	8 or 7
p = Parity	N=No Parity, E=Even Parity, O=Odd Parity
s = Stop Bits	1 or 2

Example for 300 baud, 7 Data, Even Parity, 1 Stop use:

```
RCV HELLO.BA /37E1
```

The matching parameter on the Model 100 would be

```
SAVE "COM:37E1E"
```

Note that the Model 100 uses the same string with an 'E' added on to the end to Enable XON/XOFF. The 'E' is not required for RCV.EXE because it always uses XON/XOFF protocol.

The third RCV.EXE switch displays a brief description of the command syntax and switches.

RCV /h

RCV.EXE will also display help if there is an error on the command line or switches or the specified file cannot be created.

The /N switch will cause a 'Noisy' receive by displaying a '.' dot on the screen for each full record received from the Model 100/102. Normally the RCV program displays a single asterisk '*' to indicate that reception has begun, and nothing further until reception is completed. It then flushes the capture buffer. The noisy switch may not be optimum at 9600 baud so it is normally set off.

The RCV program will abort if any key is pressed. In the event that the computer hangs up because the Model 100/102 is not sending, press any key to terminate RCV.EXE.

As mentioned before, the RCV.EXE program is very simple and receives characters by repetitively polling the COM port for characters. On some PCs, the polling is not fast enough to keep up with a 9600 baud transfer. If you experience any difficulty receiving data to the PC, then try a lower baud rate.

Receiving on the 100/102

A Description of COLOADER.EXE

Creating compiled programs on the PC presents a Catch-22 for the Model 100/102. The output file from the compile and assemble steps is an Intel HEX file. This file format is very compatible for ROM burners, but cannot be directly loaded in to the Model 100.

In order to get a .HEX file onto the 100/102 as a machine language program, a program must exist on the Model 100/102 that can receive and load this type of file. This .HEX loader should be as efficient as possible. A HEX loader could be written in BASIC, but BASIC runs very slowly when you are trying to load large files. The answer is to create a HEX loader that is written in machine language.

COLOADER.EXE handles this nicely. COLOADER is a program that runs on the PC. It will write a BASIC program that can be downloaded to the Model 100/102. The BASIC program is not the HEX loader, but when it is run on the Model 100 it will create a machine language program that is a HEX loader.

This sounds complicated, but it simple to use.

Using COLOADER.EXE

It is easiest to describe COLOADER.EXE using a real example. Assume you have created a BASIC program PROGA.BA that must be ORGed at 57000 to fit in to Model 100 memory.

The program would be compiled and assembled using that address.

```
RBASIC PROGA /o57000

A100 PROGA
```

The output from these two steps will be a file named PROGA.HEX.

Now we're ready to use COLOADER.

1. Ensure that the files COLOADER.EXE, A100.EXE and RCO.ASM are available in the current directory.

2. Calculate a second ORG address that is 500 bytes below the address of your program.

   57000 - 500 = 56500

3. Using this address, start COLOADER.EXE with the following command.

   COLOADER RCO /o56500

   This will create a BASIC program named RCO.LDR. The file is BASIC program, but does not use the standard .BA or .BAS extension.

4. Load and run RCO.BA on the Model 100. On the Model 100 type

   RUN "COM:88N1E" <ENTER>

   On the PC type

   SND RCO.LDR <ENTER>

5. While RCO.LDR is running it will display the message "Creating RCO.CO Program". This should only take a few seconds, and the 'Ok' Prompt will appear.

6. The BASIC program that was just loaded and run has now done its job and you may delete it by typing "NEW" and pressing ENTER.

7. Press F8 to return to the Model 100/102. The menu will now contain the program RCO.CO. This program is the HEX loader and can be used to load your program as described in the following steps.

8. Position the cursor over RCO.CO and press enter. The Model 100/102 Screen will clear, and the loader is now waiting for you to send your program file.

9. On the PC type

   SND PROGA.HEX <ENTER>

10. While your program is loading, RCO.CO will display dots ('.') on the screen indicating that HEX records are being received and poked into the Model 100/102 Memory.

11. When the load is complete, you will return to the Model 100/102 System Menu.

12. To start running your program, enter BASIC and type

   CALL 57000 <ENTER>

   As long as your program's original ORG address does not change, you may edit, recompile and reload using RCO.CO as many times as needed. If your program must be ORGed at a lower address because of an increase in size, then go back to Step 1 and create a new version of RCO.LDR and download and run it.

Using COLOADER.EXE with the ROM Eliminator.

COLOADER.EXE may also be used to create a loader for the SoundSight ROM Eliminator.

The loader created on the Model 100/102 is named RRM.CO and also uses about 500 bytes. If you have no other machine language programs the ORG address can be set as high as 62460 since ROM programs do not require any HIMEM.

Repeating the example of the above PROGA.HEX with the exception that it has been recompiled at address 0 to be loaded in to a ROM.

1. Ensure that the files COLOADER.EXE, A100.EXE and RRM.ASM are available in the current directory.

2. Calculate the ORG address at 500 bytes below HIMEM (in this example MAXRAM is used).

   62960 - 500 = 62460

3. Using this address, start COLOADER.EXE with the following command.

   COLOADER RRM /o62460

   This will create a BASIC program named RRM.LDR.

4. Load and run RRM.LDR on the Model 100. On the Model 100 type

   RUN "COM:88N1E" <ENTER>

   On the PC type

   SND RRM.LDR <ENTER>

5. While RRM.BA is running it will display the message "Creating RRM.CO Program". This should only take a few seconds, and the 'Ok' Prompt will appear.

6. The BASIC program that was just loaded and run has now done its job and you may delete it by typing "NEW" and pressing ENTER.

7. Press F8 to return to the Model 100/102. The menu will now contain the program RRM.CO. This program is the HEX loader and can be used to load your program as described in the following steps.

8. Position the cursor over RRM.CO and press enter. The Model 100/102 Screen will clear, and the loader is now waiting for you to send your program file.

9. On the PC type

    SND PROGA.HEX <ENTER>

10. While your program is loading, RRM.CO will display dots ('.') on the screen indicating that HEX records are being received and poked into the ROM Eliminator.

12. When the load is complete, you will return to the Model 100/102 System Menu.

13. To start running your program, enter BASIC and type

    CALL 63012 <ENTER>

Technical Information About COLOADER.EXE

If you are not familiar with assembly language programming on the Model 100/102, the following description may be of little interest to you.

COLOADER.EXE uses an Intel .HEX file as input to generate a BASIC program that contains the DATA statements necessary to poke a machine language program in to RAM along with a FOR-NEXT loop to do the poking.

Assuming that you start COLOADER with the command

COLOADER RCO

1. COLOADER opens up the HEX file RCO.HEX and reads in the bytes creating BASIC DATA statements for each byte encountered.

2. COLOADER generates a FOR-NEXT loop that starts at the ORG address and reads and pokes the data statements.

3. The last code generated by COLOADER is a SAVEM command.

   When this program is sent to the 100 and run there, it creates a machine language program by poking the program bytes into memory and then saving it as a .CO file.

   COLOADER could be used to create a loader for any Intel .HEX file or .ASM source program that can be assembled with A100, but if the ORG address is to be passed to A100 by COLOADER, then the .ASM file must not contain an ORG statement.

COLOADER will also allow an .ASM file to first be assembled to generate the .HEX file.

Assuming that you start COLOADER with the command

COLOADER RCO /o56000

the following happens.

1. COLOADER calls the A100.EXE assembler and asks it to assemble RCO.ASM using an ORG address of 56000. Aside from the forced ORG address, there is nothing special about this assembly, and A100 does its usual job of generating RCO.PRN, RCO.SYM and RCO.HEX. If you look at RCO.ASM you will notice that it does not contain an ORG address. This is deliberate so that COLOADER can pass the ORG address for A100 to use.

2. COLOADER proceeds to use the RCO.HEX file just created as input for the steps 1 through 3 of the previous example.

COLOADER also allows the name of the loader to be changed. A second file name or partial file name will be used to modify the name of the output file.

COLOADER RCO X	Use RCO.HEX to create X.LDR
COLOADER RCO A:	Use RCO.HEX to create RCO.LDR on the A: drive.
COLOADER RCO .LOD	Use RCO.HEX to create RCO.LOD.
COLOADER RCO .LOD /o59000	Assemble RCO.ASM at address 59000 and use the RCO.HEX file to create RCO.LOD.

COLOADER Switches

```
/Onnnnn The ORG Address switch.
        and
/A      The Assemble switch
```

The ORG switch /Onnnnn is used to establish the ORG address that will be passed to A100 for the assembly. The assemble switch /A is used to initiate an assembly without passing an ORG address to the A100 assembler. When using COLOADER to create RCO.LDR or RRM.LDR always pass an address as in the examples given above. You may modify or edit the files RRM.ASM and RCO.ASM but you should first keep a copy of the original in case the changes cause the loader to stop working.

If an ORG address is passed by COLOADER to A100 but the .ASM source code file already contains an ORG address, the passed address will be ignored, and the ORG statement in the file will be used. If PROGA.ASM file contains an ORG address then

```
COLOADER PROGA /o56000
        is equivalent to
COLOADER PROGA /A
```

because the passed addressed is ignored.

/N NEW Switch.

The output program generated by COLOADER ends with an END statement. After the BASIC program has poked the machine language into memory and SAVEM'd the result, the BASIC program is no longer needed.

The /N switch will cause the BASIC program to end with NEW instead of END causing it to remove itself from memory.

The NEW switch only works correctly with the poke switch, /P, described below.

/P Poke Only

Since COLOADER can be used on any .HEX file it would be possible for it to attempt to poke a large file into memory. When this happens, three areas will use up Model 100/102 memory.

1. The BASIC loader program.
2. The area of memory that the machine language loader is being poked into.
3. The SAVEM'd .CO file.

The /P switch omits the SAVEM step in the output program as one method of avoiding memory exhaustion.

/H The Help Switch

This switch provides a brief display of the syntax and switches for COLOADER. It can be used as the only argument on the command line. The help is not intended to replace the manual, but only to provide a way to jog your memory.

COLOADER /h

Displays a brief Help text about using COLOADER.

The A100 Assembler

Overview.

The A100 is a simple two pass assembler specifically designed to compile the output of the RBASIC compiler. It has a few special features to handle assembling code for the Model 100/102.

The assembler supports the complete set of 8085 opcodes including SIM and RIM and a subset of the common pseudo-ops.

It does not support math operators or macros.

OPCODES - This is the standard set of Intel mnemonics for the 8085 chip. The 8080 set plus SIM and RIM. I refer you to any standard text on these.

OPERANDS - Operands may be decimal, hex or labels. Hex numbers must start with a digit, and be terminated with H or h. Labels must be defined somewhere within the source module being assembled. Operands may not contain math operators. The following are not legal.

```
        LXI     H,5302          ;Legal Decimal Value
        SHLD    0AF70H          ;Legal Hex Value
        LDA     BUFFER

        LDA     5302+16         ;Illegal Math Operator
        LXI     D,2405-8        ;Illegal Math Operator
```

LABELS - Labels must start in column 1 and must end with a colon. Only the first 8 characters are significant.

```
DOIT:
        MOV      A,1
        JMP      DOIT2
DOIT1:
        DB       5
DOIT2:
        JMP      SOMEWHERE
```

COMMENTS - Comments must be preceded by a semicolon. Everything after the semicolon is ignored.

```
;This is a comment
        XRA      A         ;And so is this
```

PSEUDO-OPS - Several standard and a few special pseudo-ops are supported.

```
CTRLZ:  EQU      26        ;EQU to assign a value to a label

MSG:    DB       65,66,0   ;DB to create bytes of storage
    or
MSG:    DB       'ABC',0

TABLE:  DW       5507      ;DW to create words of storage
        DW       2205,0

BUF:    DS       256       ;DS to reserve storage
```

A storage area reserved with DS is not filled in by the assembler and simply causes the current position counter to skip the specified number of bytes.

INCLUDE - Include causes another file to be included at this point of the assembly. The file is included and assembled in line as if the original code had been written into the file being assembled.

```
        INCLUDE  SOURCE.ASM
        INCLUDE  IO.SUP
```

ROM - ROM is a special pseudo-op that switches the assembler to ROM mode. It controls the way the compiler treats the special opcodes covered in the next section. It has no parameters and is usually included at the beginning of the source code file so that it affects the entire assembly.

Special Opcodes.

The A100 assembler supports three special Opcodes that are not a standard part of the 8085 command set. They were added to the assembler for ROM development. There are two mnemonics for each for each of the three opcodes.

The STDROM or CSR opcode is similar to a CALL but it is specifically used for CALLing a routine in the Model 100/102 Standard ROM.

The STDJMP or JSR opcode is similar to a JMP but it is specifically used for jumping to a routine in the Model 100/102 Standard ROM.

The STDRET or RSR opcode is similar to a RETurn but is used to return from an Option ROM routine called by a Standard ROM routine.

```
        STDROM   4b44H    ;Call print char routine
or      CSR      4b44H

        STDJMP   5797H    ;Jmp to menu routine
or      JSR      5797H

        STDRET            ;Return to Standard ROM.
or      RSR
```

These three instructions, in either variation, assemble differently depending on whether the ROM pseudo-op has been set. When the ROM PSEUDO-OP is not set, they assemble as standard CALL, JMP and RET instructions. When the ROM PSEUDO-OP has been used they assemble as indirect CALL, JMP and RET instructions executing via the code in the ROMHEAD.SUP module.

Assembler Output Files

The output from A100 is three files. The .SYM file contains a list of all the labels in the .ASM file and the values that were resolved for them at assembly time.

The .PRN file contains a complete listing of the assembled file including any files that pulled in to the main module with the INCLUDE directive.

The .HEX file is an Intel hex format file suitable for loading to a Model 100/102 using RCO.CO or RRM.CO. The format is used by nearly all ROM burners on the market.

Compiling for MAC vs A100.

There are really only two differences in the output files generated by RBASIC when the /M switch is used to generate a file that is compatible with the Digital Research Assembler.

MAC does not support the INCLUDE pseudo-op, so the compiler actually physically copies INCLUDE files into the output .ASM file and does not use the INCLUDE directive.

MAC also does not support the specially opcodes STDROM, STDJMP and STDRET. Instead the compiler generates macros at the start of the .ASM file that MAC can handle. In fact the three special opcodes are really more like macros, but they have been embedded in A100 whereas they must be defined as macros for MAC.

ON ERROR Processing

Using No Error Checking

ON ERROR processing capability has been included in RBASIC but not without a cost in space in the final output.

The lowest level of error processing is none at all. RBASIC requires that a line number be stored for each line that is executing. Without any checking, the default line number 9999 is loaded and saved once to be used as the line number throughout the program. If an error occurs you will be dropped into the BASIC error screen with the standard error message specifying a line of 9999

?SN Error in 9999

If the code has been thoroughly checked and verified, you may not care about this because if no errors come up this will never present a problem.

Errors Using Line Numbers.

The next step up from here is to include line numbers at compile time using the /L switch. This will cause the compiler to generate code that will load and save each line number at the start of the line. If an error occurs under these conditions the line number will be the actual line in which the error occurred. (Please see the one exception on FOR-NEXT loops in the Chapter on RBASIC syntax).

You will still drop into the BASIC Ready screen.

?SN Error in 510

This may be a wise move during development as line numbers will help you locate and iron out errors.

ON ERROR Logic.

ON ERROR logic uses up even more code space. For every line the line number must be saved, and for every statement the addresses of the current statement and the next statement must be saved. The address of the current statement is used for a RESUME command, and the address of the next statement is used for a RESUME NEXT command.

This is not much of a problem for ROM Development since you usually have room to spare in a ROM, but it can cause problems for .CO development.

ON ERROR Space Saving Tips

The compiler has been designed to allow you to localize your ON ERROR logic if you only need it for a portion of the code, but not for all.

A classic example is dealing with legal errors. The favored legal error is trying to open a file input to determine if exists and trapping the error if any.

```
10 EF=0:GOSUB 500
20 IF EF THEN PRINT "FILE NOT FOUND"

500 ON ERROR GOTO 600
510 OPEN "FILE.DO" FOR INPUT AS 1
520 ON ERROR GOTO 0
530 RETURN

600 EF=1:RESUME NEXT
```

The samples set a flag to zero and then execute a subroutine to open the file. The subroutine sets an error exit that will set the flag to one, attempts to open the file and then clears the error trap. The error trap if it is executed will set the flag to 1 and this condition is tested at line 20.

In the above example the only error that is of concern to the programmer is the event of a file not found. Any other error that occurs is going to drop out of the program into the standard BASIC error screen. If you have a situation like this, you can greatly optimize the code by placing the subroutine at 500 at the very end of the code.

If the compiler is run without the error trapping switch /E, it will not generate any error trapping code until it encounters the first ON ERROR statement at line 500. The compiler will display "Warning ON ERROR logic implemented at line 500" but since that is the only place you need the logic it presents no problem.

If you still wanted to trap correct line numbers in the event of some other error use the /L switch on the compiler.

```
RBASIC PROGA /L
```

This would give you line numbers for all of your program, and ON ERROR logic only at the end of the program for the one routine that needs it.

ON INTERRUPT Processing

Overview

ON INTERRUPT Processing is implemented in RBASIC in approximately the same way it is done for interpreted BASIC. Before each statement is executed a check is made to determine if an interrupt event has occurred. If one has and the interrupt is enabled, that interrupt is serviced before the statement is executed.

Normally the /I switch is used to tell the compiler that ON INTERRUPT logic must be compiled in to the code.

Space Saving Tips

As with ON ERROR logic, the routines that actually require interrupt processing can be moved to the bottom of the code and if the switch is not used at compile time, the compiler will not generate ON INTERRUPT checking logic until it encounters the first ON COM, ON TIME$, or ON KEY command.

For example a screen routine that needed to intercept ON KEY interrupts only while the screen was being displayed might look something like this.

```
100 ON KEY GOSUB 200,300,400,500,600,700,800,900
110 GOSUB 1000 'Display all prompts
120 GOSUB 2000 'Enter all fields
130 KEY OFF:RETURN
```

The routines at 100,1000 and 2000 could all be moved down at clumped at the bottom of the code. The routines at 200 through 900 need not be moved unless they themselves are expecting to do additional interrupt handling.

The basic rule is that routines that may be interrupted should appear after the first ON INTERRUPT instruction. The ON INTERRUPT instruction should appear as late in the code as possible. With this done compile without the /I switch and the compiler will turn on interrupt processing logic only for the portion of code after the ON INTERRUPT command.

Again the compiler will display a warning such as "Warning ON INTERRUPT Logic implemented at Line 4000".

Compiler Error Messages

There are few compiler error messages. The RBASIC compiler is designed around the idea that the program was alive and well as an interpreted BASIC program and does not do a lot of syntax checking.

Keyword Errors

The keyword in a BASIC statement is the first token that indicates that an action is to be performed. PRINT, IF, LET (stated or implied) GOSUB, GOTO, ON are all examples of keywords.

MERGE not supported in line nnnnnn.
This error is displayed when any unsupported keyword is encountered. The unsupported keywords are listed in the chapter on RBASIC Syntax. If you get this error, remove the offending statement. RBASIC will not compile it.

Missing Keyword in line nnnnnn.
This error is displayed when a statement exists without a keyword such as
20 "Hello"
If you get this error correct the code to include the right verb.

Invalid Keyword in line nnnnnn.
This error indicates that a statement starts with a BASIC token that is not legal as a verb.

```
30 SGN(A)
```

In the above example SGN is a legal BASIC token, but it may not be used as at the start of a statement. The correct syntax would be LET B=SGN(A).
If you get this error correct the line and recompile.

Target Line Errors

Invalid GOTO or GOSUB Target in line nnnnnn.
This error indicates that a GOTO or GOSUB contains a missing or invalid target line such as attempting to GOTO or GOSUB a variable (which some BASICs do allow).

```
30 GOSUB AX
```

Note that the compiler does not produce errors when the target of a GOTO or GOSUB does not exist. The Line

```
30 GOTO 500
```

will generate the assembly language instruction

```
        JMP      L500
```

If line 500 does not exist, the compiler does not know this. The assembler will pick it up and display an error message of
UNDEFINED Label L500

IF ELSE Errors

IF ELSE Too Complicated at line nnnnnn.
The compiler internal tables allow 10 levels of IF-ELSE nesting within a single statement. If the statement contains more than this the above message will be displayed. To correct this error, break up the IF-ELSE into smaller lines.

FOR NEXT Errors.

Missing or Invalid Variable in NEXT in line nnnnnn.
This error indicates that a NEXT command has been issued without naming the variable, or a NEXT has been issued for a variable that is not in use in a FOR statement. If you get this error, add or correct the NEXT variable.

```
10 FOR A=1 to 10
20 NEXT B
```

or

```
10 FOR A=1 to 10
20 NEXT
```

Syntax Errors

Syntax Error in line nnnnnn.
This is general purpose catch all error. If you see this error, take a look at the code and correct whatever is wrong. If the syntax is acceptable to BASIC it should work for RBASIC with a few exceptions to do with undocumented syntax covered in the chapter on RBASIC syntax.

Assembler Errors

Unless you hand modify the assembler output code from the compiler, the only error message that you will encounter will be

Undefined Label Lnnnn

where nnnn will be decimal number as in

Undefined Label L9070

The RBASIC compiler creates labels for line numbers by preceding the line number with an 'L'.

The above error indicates that the original BASIC program contains a GOSUB or GOTO 9070, but line 9070 does not exist. This is the one error that is picked up at assembly time rather than compile time.

The assembler also provides important information about the final output code.

The main piece of information is the End address of the .HEX file. When you are assembling for an Option ROM, this address can help to decide the size of ROM needed.

```
0000H    -        1FFFH    =        8K ROM
2000H    -        3FFFH    =       16K ROM
4000H    -        7FFFH    =       32K ROM
above 7FFFH will not fit on a ROM
```

The assembler assumes that you are developing for the Model 100/102. If the org address is set higher than 0000H which is the start address for a ROM, the assembler operates on the basis that you are assembling a file intended to be loaded as a .CO file. If the End address is higher than the Model 100/102 MAXRAM (62960 decimal or F5F0 Hex) it will display a warning message that the .HEX file will not load and run as a .CO program because it will overwrite reserved system memory during the load.

Additionally for non-ROM assemblies (assembled at an ORG address other than 0000) the assembler displays the highest possible ORG address that can be used for this program. If you have received the warning message that your code has extended above F5F0, then you must use this recommended ORG address or a lower one if you want your program to load and run as a .CO file on a Model 100/102.

Technical Information

Compiler Description.

The output of the RBASIC compiler uses a great deal of the BASIC interpreter to get code executed.

The code that is generated contains a tokenized line that looks almost exactly like the original line with out the key word. For example

```
10 PRINT A,B
```

will compile down to a tokenized line containing 'A,B' and a call to the Model 100/102 PRINT routine. This call is done indirectly through a support routine that will be called XPRINT.

When the code is executed the following steps are done:

1. If the code is being generated for an Option ROM, the tokenized line is copied to a buffer in RAM (see reserved memory below).

2. The HL register is loaded to point to the first character of the tokenized line either in the code or in the reserved buffer.

3. The Corresponding Model 100/102 routine is called directly (for .CO code) or indirectly (for ROM code).

   These steps are so straight forward for the bulk of the BASIC verbs that the code is generated directly in the compiler.

There are a few verbs such as READ where this process will not work and some additional processing is necessary before the interpreter can take over. For these, special support code is written and included in a support file such as XREAD.SUP

A third category exists in which the interpreter cannot be used at all such as in tracing. The XTRACE.SUP support file is a complete trace support routine. It calls routines in the standard ROM to do such things as display the line numbers.

A fourth category of verbs present straight faster compiled versions.

```
GOSUB 100 becomes          CALL     L100
GOTO  100 becomes          JMP      L100
```

You can obtain a fair understanding of the compiler output by compiling a program with the /C (include lines as comments) option and then assembling it. Print or display the resulting .PRN file which will incorporate all INCLUDE files and should give you a good idea of how the process is being done.

Reserved Memory.

The output code from the compiler requires a buffer in RAM. Rather than require that the program clear HIMEM before it can be run, the TELCOM back page is used as a buffer. This is an are of memory that extends from FCC0H to FDFFH (64704 through 65023). If your program uses any of this memory it will have to be changed to place your data somewhere else. This area is frequently referred to as the TELCOM back page or the ALT-LCD buffer. It is normally used by TELCOM to store the previous page of data.

Code Execution Speed

Although speed was a secondary issue in the design of the compiler, there are some areas in which the compiler generates code that is faster at execution time.

1. Keywords. Every time the interpreter begins a new statement a great deal of set up code is executed, then the interpreter goes through a dispatching process to identify the key word and jump to the correct routine. In RBASIC the set up code has been minimized and the compiler generates code that directly calls the keyword to be executed.

2. GOSUB-GOTO. When the interpreter encounters a GOSUB or GOTO, it must evaluate the following line number and then search through the BASIC program to locate the line before transferring control to it. The further away the line is in the code, the longer it takes to find it. In RBASIC the line numbers are already known and located. A GOSUB translates to a CALL and GOTO translates to a JMP. The transfer time is identical no matter where the line is.

One area where RBASIC is slower is in string assignments. All string assignments done with a LET or READ command must be converted to string operations for ROM code.

```
10 LET A$="HELLO"
becomes
10 LET A$="HELLO"+""
```

```
10 READ A$
20 DATA "HELLO"
becomes
10 LET A$="HELLO"+""
```

This is designed to force the string variable to be moved into the string space variable area where it can be accessed, It cannot be directly accessed in the option ROM and this will slow down assignment and READ operations on strings.

Overwriting HIMEM.

A common error that can occur during testing is ORGing the program at an address that is too high in memory.

The A100 assembler displays the highest address of the code on the screen as

End address is 9d27=40231d

If the hex value is higher that F5F0 or the decimal value is higher than 62960, a warning is displayed and the program should be re-compiled with a lower ORG address.

If any attempt is made to load the above file some of it will be poked in to reserved system memory and the Model 100/102 will crash during the download.

Questions and Problems

The following is a list of common questions and problems that have come up using the RBASIC system.

1. Can more than one BASIC program be compiled in to one ROM?

   Under Version 5.0 of the RBASIC Compiler, yes. (See Chapter 16)
   Under version 4.0, the simple answer is no. This version of the compiler is not designed to handle this. It would be possible for someone well versed in Assembly language to compile two or more programs and perform some deft surgery on the .ASM files to get them all in to one ROM, but it is not an easy task. It would probably be simpler to combine all the programs in to one and include a BASIC front end menu that jumps to the appropriate part of the complete program.

2. While downloading a .HEX file to be tested as a .CO file before burning in to a ROM, the Model 100/102 locks up?

   You are probably ORGing the file too high in memory and overwriting reserved system memory during the download. See the section on Overwriting HIMEM in the chapter on Technical Information.

Known Bugs

Please refer to this list, and the list of Syntax problems in the chapter on Syntax before reporting a bug. They may already be known about.

1. When Line numbering or tracing is used, RBASIC can loose track of the correct line number within a FOR-NEXT control.

```
10 FOR A = 1 to 10 : STATEMENT1:STATEMENT2:
20 NEXT A
```

The NEXT at line 20 forces control back to STATEMENT1 then STATEMENT2. When control is forced back to STATEMENT1, the line number is not reset to 10, but remains 20. If an error occurred in STATEMENT1 or STATEMENT2, the error message would indicate an error in line 20. If certainty is required on line numbers within a control statement, then the first statement after a FOR must start on its own line number. In practice this is rarely a problem.

```
10 FOR A = 1 to 10
20 STATEMENT1:STATEMENT2:
30 NEXT A
```

2. If an error occurs when the program is already executing an error trap routine, normally the error should cause the program to drop into a BASIC error screen and stop running.

```
10 ON ERROR GOTO 100
20 PRNT "HELLO"
30 END
100 PRNT "TRAPPED THE ERROR"
110 RESUME NEXT
```

In the above example the error at line 20 would cause the routine at 100 to be called. The error in line 100 will cause a program abort and syntax message.

```
? SN Error in line 100
```

Under certain circumstances which have not yet been isolated, RBASIC will get caught in an endless loop. The error at line 100 causes another call to 100. This can only be undone by pressing reset.

The only solution is to ensure that your error trapping routines do not themselves contain errors or the potential for error.

3. The Slow Trace switch does not work on lines that use an INKEY$ loop.

```
100 A$=INKEY$:IF A$="" THEN GOTO 100
```

If a program containing the above line were compiled with the /S option, the program would hang up at line 100 when run. The problem is a conflict between the key needed for the slow trace, and the key needed for INKEY$. It is nearly impossible to press two keys fast enough to respond to the slow trace and then provide a key that can be picked up by INKEY$.

If a slow trace is needed, then change the line to an INPUT statement for the debugging session and resign yourself to pressing ENTER after the keystroke.

```
100 INPUT A$
```

4. As of this writing, there has been one report of a problem with the LOADM command producing a Syntax Error. The report is not clear whether the error is occurring during compilation or at run time. We have as yet been unable to duplicate the problem and would appreciate hearing from anyone who runs in to this one.

A LOADM command will not work within a subroutine but that is true for both RBASIC and interpreted BASIC. The program misbehaves differently but for the same reason.

```
10 GOSUB 100
20 PRINT "Load Completed"
30 END
100 LOADM "PROG.CO"
110 RETURN
```

The LOADM at line 100 destroys the stack for either BASIC or RBASIC.

In BASIC this produces the error

? RG Error in line 110 (RETURN without GOSUB Error)

In RBASIC the program exits to the Model 100 Menu, or produces a spurious error because the RETURN has lost its place in the stack and will jump to some unknown piece of code (usually the Menu).

For LOADM to work within RBASIC, the command must not be issued within a subroutine.

5. Using the Radio Shack bar code package has revealed a bug in the RBASIC verrsion of RUNM. This Bar Code package includes three drivers for PLESSY, UPC or 3-OF-9 code. The recommended procedure in the manual is to RUNM "file name" such as

```
RUNM "B3OF9"
```

Then OPEN the WAND for input. In RBASIC this is producing spurious error messages. The same effect can be achieved by using LOADM and CALL which is functionally equivalent to RUNM.

```
LOADM "B3OF9":CALL 61824
```

This bug will show up in other efforts to run or use machine language routines with a BASIC program.

In general terms you can get around the RUNM problem by loading the program using LOADM and then CALLing the correct entry address. If you do not know the correct entry address then use the LOADM command by itself to establish this. If you had a file called DOIT.CO on your Model 100/102 menu and did not know the entry address for it, you could enter BASIC and type

```
LOADM "DOIT"
```

This might display

```
Top  :  59123
End  :  62950
Exe. :  59555
```

The address listed as the Exe. address would be used as the CALL address to get around the bug in the RBASIC version of DOIT. This would be coded as

```
10 LOADM "DOIT":CALL 59555
```

The Top address is the highest value that HIMEM may have that still allows DOIT to run. The RBASIC program that runs DOIT must contain a clear statement to set HIMEM to 59123 or lower statement such as

```
CLEAR 100,59123
```

6. The RBASIC compiler will create a ROM trigger file name on the menu if requested by using the /R switch.

```
RBASIC DODAH /RDODAH
```

The above command will create an entry on the Model 100 menu of "DODAH" that can be used to start the program.

This file cannot be correctly killed. The command

```
KILL "DODAH"
```

will result in the Model 100/102 internal counters being messed up and the Menu will display a free bytes count in excess of 60000 bytes.

7. The Compiler does not correctly handle MAXRAM in a clear statement. As in

```
10 CLEAR 500,MAXRAM
```

This must be changed to

```
10 CLEAR 500,62960
```

8. The compiler generates a Missing KEYWORD Error when a space precedes an ELSE after a COLON.

```
100 IF A=15 THEN PRINT A: PRINT B: ELSE PRINT C
                                  ^
```

This space causes a missing KEYWORD Error. Correct this by eliminating the space.

```
100 IF A=15 THEN PRINT A: PRINT B:ELSE PRINT C
```

---

9. A bug exists in the PRINT USING logic under certain circumstances when using PRINT USING with a variable.

```
10 O$="###.##":PRINT USING O$;123.5:B$="Hello"
```

The above line causes the value "Hello" to be assigned to to O$ as well as B$.

A similar problem happens when using a PRINT STRING$() af ter a PRINT USING.

```
20 O$="###.##":PRINT USING O$;123.5:? STRING(5,49)
```

This casues O$ to be set equal to STRING$(4,49) or in this case "11111".

The reason is not known, but a fix is being worked on. The bug has always existed in the compiler but was not previously discovered because of the unusual combination of conditions.

1. A PRINT USING using a variable.
2. Immediately followed by an assigment or PRINT STRING$() command.

For the time being, you can code around this problem by immediately reassigning the variable to itself after the PRINT USING statement.

The fixes for the above two lines would read

```
10 O$="###.##":PRINT USING O$;123.5:O$=O$:B$="Hello"

20 O$="###.##":PRINT USING O$;123.5:O$=O$:? STRING(5,49)
```

Assembly Language MERGE

Extending BASIC.

The MERGE verb cannot be used in RBASIC. The program once compiled is immutable, therefore another BASIC source file cannot added to the code.

Rather than let a perfectly good verb go to waste, a special extension to the language has been implemented starting with version 2.00 of RBASIC.

The MERGE verb may be used to include an assembly language source code file directly into the RBASIC program. This will primarily be of benefit to assembly language programmers and the coverage of this subject assumes that you have some knowledge in this area.

Assembly Source Compatibility

RBASIC will generate .ASM files that can be assembled with A100, or the CPM MAC assembler. Any assembly language code must be written so that it will assemble correctly. Please see the section on A100 for a brief rundown on the assembler.

The intended use of the MERGE command is to allow routines that must be fast to be included and directly callable from within the RBASIC program.

Overview of Techniques.

The technique is fairly straight forward.

1. Write a assembly language routine and place it in a text file.

2. Place a MERGE command in the RBASIC source code file that names the text file.

3. Compile and assemble as before.

For example a routine could be written and placed in a file called HOOPLA.ASM. Line 500 below would cause the contents of HOOPLA.ASM to be added to the output generated by the RBASIC compiler.

```
10  GOSUB 500 or GOTO 500
500 MERGE HOOPLA.ASM
```

The entire routine in HOOPLA.ASM would have the line 500 label L500 and GOSUB 500 at line 10 would actually cause a call to the L500 label.

```
L10:
        CALL     L500
```

The entire output file which includes the BASIC program and the logic from HOOPLA.ASM would then be assembled by A100.EXE.

Labeling Restrictions.

It is good practice to avoid using labels in a MERGE file that might conflict with labels generated by RBASIC. RBASIC generates three basic types of labels within the program logic.

Line labels use the original program line number preceded by an 'L'. Line 500 becomes L500, line 2705 becomes L2705 and so on.

Statement labels are used to mark the beginning of statements. Statement labels begin with an X and then contain digits. Statement numbers are generated during the compile sequentially. The first statement label will be X1, the next X2 and so on to the end of the program.

Jump labels are generated in a similar way and are used to mark jumps for IF-ELSE and FOR-NEXT logic. Jump labels also start with X followed by digits.

Outside the program logic, support routines are generated usually starting with an X. For example the PRINT support routine is called XPRINT, and READ support routine is called XREAD.

The last set of labels are actual addresses within the Model 100/102 operating system. They are all listed in SYSAD.100.

If you avoid using these names as labels, you can create a file that can be directly inserted in a program.

Program Control Restrictions

The program control restrictions are pretty much what you would expect.

1. If a routine is inserted at a line number that is called with a GOSUB, then it must end with RETurn.

2. If a routine is inserted at a line number that is jumped to with a GOTO then it must NOT end with a RETurn.

3. If a routine is inserted at a line number that is executed in line in the code, then it must NOT end with a RETurn.

4. The routine must not alter the stack, and should not change any BASIC variables (as named in SYSAD.100) unless you have carefully studied what the program is doing.

5. If the routine will be calling a Standard ROM routine then use the CSR or STDROM macro to ensure that the CALL will always assemble correctly.

```
        STDROM   4b44h              ;Display one character
or
        CSR      4b44h              ;Display one Character
```

instead of

```
        CALL     4b44h              ;Incorrect
```

Programming Examples.

UPPER.ASM is routine that locates the BASIC variable CS$ and converts the entire contents of the string to upper case.

```
;-----------------------------------------------------------
;UPPER.ASM
;Locate the Variable CS$ and convert its contents to upper
;case.
;-----------------------------------------------------------
UPPER:
        LXI     H,CASEVR            ;Locate variable 'CS$'
        STDROM  4790H               ;VARPTR in Standard ROM
        MOV     A,D                 ;Ptr to Varptr in DE
        ORA     E
        RZ                          ;Return if no Var found
        LDAX    D                   ;LENGTH
        ORA     A                   ;Return if Zero length
        RZ
        MOV     C,A                 ;Length to C
        INX     D
        XCHG                        ;PTR to STR addr in HL
        MOV     E,M                 ;PTR to string in DE
        INX     H
        MOV     D,M
        XCHG                        ;Now in HL
UP1:
        MOV     A,M
        CPI     'a'                 ;If less than 'a'
        JC      UP2                 ;Skip
        CPI     '{'                 ;If gt or = 'z'+1
        JNC     UP2                 ;Skip
        ANI     0DFH                ;Turn Off Bit 5
        MOV     M,A
UP2:
        INX     H                   ;Step through string
        DCR     C
        JNZ     UP1
        RET                         ;RET for use with GOSUB
CASEVR:
        DB      'CS$'
```

This routine is set up to be called in an RBASIC program. The following program accepts user input, converts it to upper case, and then re-displays the results. The variable CS$ has been permanently reserved as the variable to use for case conversions. The routine is accessed, by setting CS$ = to the string to convert and then doing a GOSUB to the line number for the routine.

```
10 PRINT "Input A STRING";
20 INPUT X$
30 CS$=X$
40 GOSUB 1000
50 PRINT "The Converted string is"
60 PRINT CS$
70 END
1000 MERGE UPPER.ASM
```

A similar routine to do lower case conversion could be included in the same program.

```
10 PRINT "Input A STRING";
20 INPUT X$
30 CS$=X$
40 GOSUB 1000
50 PRINT "The Upper Case string is ";CS$
60 GOSUB 2000
70 PRINT "The Lower Case string is ";CS$
80 END
1000 MERGE UPPER.ASM
2000 MERGE LOWER.ASM
```

```
;-----------------------------------------------------------
;LOWER.ASM
;Locate the Variable CS$ and convert its contents to lower
;case.
;-----------------------------------------------------------
LOWER:
        LXI     H,CASEVR        ;Locate variable 'CS$'
        STDROM  4790H           ;VARPTR in Standard ROM
        MOV     A,D             ;Ptr to Varptr in DE
        ORA     E
        RZ                      ;Return if no Var found
        LDAX    D               ;LENGTH
        ORA     A               ;Return if Zero length
        RZ
        MOV     C,A             ;Length to C
        INX     D
        XCHG                    ;PTR to STR addr in HL
        MOV     E,M             ;PTR to string in DE
        INX     H
        MOV     D,M
        XCHG                    ;Now in HL
LW1:
        MOV     A,M
        CPI     'A'             ;If less than 'A'
        JC      LW2             ;Skip
        CPI     '['             ;If gt or = 'Z'+1
        JNC     LW2             ;Skip
        ORI     020H            ;Turn On Bit 5
        MOV     M,A
LW2:
        INX     H               ;Step through string
        DCR     C
        JNZ     LW1
        RET                     ;RET for use with GOSUB
```

Note in the above examples that CASEVR only exists in UPPER.ASM, but not in LOWER.ASM. Having it in both would cause a duplicate name conflict so one file must rely on the other to have the data named.

It would also be possible to place CASEVR in a third file and include it as a single 3 byte data field. As long as the line number to which it was attached was never executed, it would hold data without any problem. In the example below, CASEVR would have to be removed from UPPER.ASM.

```
;CASEVR.ASM

CASEVR: DB       'CS$'

10 PRINT "Input A STRING";
20 INPUT X$
30 CS$=X$
40 GOSUB 1000
50 PRINT "The Upper Case string is ";CS$
60 GOSUB 2000
70 PRINT "The Lower Case string is ";CS$
80 END
1000 MERGE UPPER.ASM
2000 MERGE LOWER.ASM
3000 MERGE CASEVR.ASM
```

Assembly language routines can call one another provided the routines RETurn. Because the UPPER.ASM file starts with the label UPPER:, the routine included at line 1000 will have two labels, L1000 and UPPER. LOWER.ASM or any other assembly language source code file could access the UPPER routine with

```
        CALL     L1000
```

or

```
        CALL     UPPER
```

Obviously the second choice makes more sense since the file could be included at any line number.

You can also call or jump to a BASIC line number from within the assembly language routine provided the rules for program control are followed.

It would be possible to write a machine language sorting routine that called BASIC program lines to display progress messages so that the machine language logic could be kept simple.

Option ROM Considerations.

In the previous examples, a constant value for 'CS$' has been created in the assembly language routines to be accessed by the VARPTR routine.

Once the program is moved to an Option ROM, accessing this memory becomes more awkward.

First the memory cannot be used to store data. It will be burned into the ROM and will not be modifiable. Second, routines in the Standard ROM cannot be passed pointers to data stored in an Option ROM. Standard ROM routines cannot 'point' into the option ROM.

This problem was overcome in compiler design by copying data to a buffer in the TELCOM back page area (FCC0H) and then processing the data in that buffer. There is no reason that you cannot take advantage of the logic.

The routine is called CP2BUF, and the address of the buffer is SUPBUF. The CP2BUF routine expects the data to be null terminated. Set HL to point to the data and CALL CP2BUF. On exit the DE register pair contain the address of SUPBUF.

Given these parameters, the following is an example of UPPER.ASM modified to run in an Option ROM.

```
;------------------------------------------------------------
;UPPER.ASM
;Locate the Variable CS$ and convert its contents to upper
;case. Runs as CO or ROM routine
;------------------------------------------------------------
UPPER:
        LXI     H,CASEVR        ;Locate variable 'CS$'
        CALL    CP2BUF          ;Copy up to RAM
        XCHG                    ;Point HL at the copied
                                ;data
        STDROM  4790H           ;VARPTR in Standard ROM
        MOV     A,D             ;Ptr to Varptr in DE
        ORA     E
        RZ                      ;Return if no Var found
        LDAX    D               ;LENGTH
        ORA     A               ;Return if Zero length
        RZ
        MOV     C,A             ;Length to C
        INX     D
        XCHG                    ;PTR to STR addr in HL
        MOV     E,M             ;PTR to string in DE
        INX     H
        MOV     D,M
        XCHG                    ;Now in HL
UP1:
        MOV     A,M
        CPI     'a'             ;If less than 'a'
        JC      UP2             ;Skip
        CPI     '{'             ;If gt or = 'z'+1
        JNC     UP2             ;Skip
        ANI     0DFH            ;Turn Off Bit 5
        MOV     M,A
UP2:
        INX     H               ;Step through string
        DCR     C
        JNZ     UP1
        RET                     ;RET for use with GOSUB
CASEVR:
        DB      'CS$',0         ;Add null for CP2BUF
```

Multiple Programs on One ROM

APPLIES TO VERSION 5.0 ONLY

Multiple Programs

It is possible to include multiple BASIC programs on one ROM, but it may be complicated.

The main problems are brought about by inter-program conflicts. Many programs initialize various values, and rely on the exit or END of the program, or the next RUN statement to re-initialize these.

For example it is possible for a program to set up ON KEY interrupt logic. When the program exits or ENDS it is not necessary to clear these interrupts as the END statement and or next RUN statement will clear the interrupts.

If multiple programs are merged into one ROM, an exit does not occur between programs. Moving from program A to program B leaving interrupts installed that were set in program A can cause very hairy results.

It is important to remember these facts as a program that runs fine on its own may leave things set up that interfere with other programs.

Parent and Child.

In order to allow the creation of multiple programs on a ROM it is necessary for one program to be named as the parent program. This would usually be a menu type program that dispatches to all the other programs.

All the remaining programs on the ROM are classed as children or descendant programs.

Inheritance.

Parent and child process will pass inherited characteristics back and forth even so far as one child program passing a characteristic back through the parent process to another child program.

The parent should be careful to set the correct characteristics for each child, or the child should reverse any system wide parameters that it sets while it is running. A common close for a child process would be to close all files, clear all variables and return to the parent.

```
100 DEFDBL A-Z:CLOSE:CLEAR:RETURN
```

Other areas that may need reversal are interrupts, screens, and function key labels which may require a more elaborate close.

```
100 SCREEN 0,0: COM OFF: ON COM GOSUB 0
105 KEY OFF: ON KEY GOSUB 0,0,0,0,0,0,0,0
110 GOSUB 200 'Routine to reset Function Keys
120 DEFDBL A-Z:CLOSE:CLEAR:RETURN
```

It may require some experimentation to determine what is affecting other programs in the family.

Table 16-1 includes a list of inheritance.

Table 16-1 Parent-Child Inheritance

P->C Parent will pass on to child
C->P Child will Pass back to parent
C->C Child will pass through parent to another child.

Action	P->C	C->P	C->C
COM Interrupts	T	T	T
MDM Interrupts	T	T	T
KEY Interrupts	T	T	T
OPEN FILES	F	T	F
Variables	F	T	F
SCREEN	T	T	T
Label Enable	F	T	F
FKey Defs	F	T	F
String Usage	F	T	F
DEFs	T	T	T
DIMs	F	T	F

Differences in a Child Program.

A child program differs from a normal RBASIC program in several areas.

1. It should exit with a RETURN verb instead of MENU or END or STOP. The RETURN will cause a RETURN to the PARENT process.

2. It can pass characteristics back to the parent.

3. A child process is always compiled with no ORG address.

4. A child process must use a labeling method that avoids clashing with the labels used in the parent or other child programs in the system. The child/descendant switch must be followed by a single letter used to modify the standard labels that are generated by the compiler. For example if a system were to include three child programs, each would be compiled with a separate letter.

RBASIC PROGA /DA   Descendant using 'A' for labels

RBASIC PROGX /DB   Descendant using 'B' for labels

RBASIC PROGA /DC   Descendant using 'C' for labels

The alpha letter must be unique among all the children to be included on one ROM. This limits a ROM to 26 children using A through Z.

4. The code of a child program may include

```
MAXFILES = nn
CLEAR string space
        and/or
CLEAR ,himem
```

but they are all ignored. A child program assumes that MAXFILES, string space and himem are all taken care of by the parent. In developing a multiple program ROM, take the highest of all the MAXFILES, the highest string space and the lowest HIMEM of all the child programs, and use those values in the parent program.

5. When a child program is compiled none of the support code is added to the .ASM file.

6. A child program is not assembled separately. All child .ASM files are intended to be used in a MERGE verb in the parent program.

7. A child program must exit at the correct level. A child program acts as if it has been started by a GOSUB, and the RETURN to get back to the parent must be located as if the whole child program were just a subroutine.

Differences in Parent Programs.

A parent program is very similar to a normal RBASIC program with a few simple differences.

1. The parent should include a MAXFILES, and CLEAR statement that encompass all the MAXFILES and CLEAR statements of all the children.

2. The parent will include all the support code for all routines. Usually an RBASIC program only includes support code for routines that are used within the body of the program. Since an RBASIC parent has no way of knowing what support the children will need, it includes all routines.

3. The parent will call any child program by using the MERGE and GOSUB technique described in the chapter on MERGE.

4. A parent can inherit characteristics from the most recently run child.

The listing of MENU.BA illustrates a parent program that calls one of three child programs, FILER.BA, GCM.BA or SYSDAT.BA.

The three children are MERGED at lines 1000, 2000 and 3000 as GCM.ASM, SYSDAT.ASM and FILER.ASM. The program has been constructed to allow for 7 programs to be run from the menu by pressing Function keys.

MENU.BA would be compiled with a parent switch as follows

RBASIC MENU /P (other switches)

Before this could be done, all three of the children processes would have to be compiled.

SYSDAT.BA is the listing for one of the three children. Note at line 60 that the program exits by using RETURN rather than END.

Each of the three programs would have to be compiled with a separate letter to allow the labels to be created uniquely.

```
RBASIC /DA SYSDAT [other switches]

RBASIC /DB FILER  [other switches]

RBASIC /DC GCM    [other switches]
```

Once these three have compiled successfully the parent program MENU.BA can be compiled and assembled.

MENU.BA

```
10 'Simple Parent MENU
20 CLEAR 1000 'Use largest values of all children
25 MAXFILES=4 'Ditto
30 GOSUB 200 'Init and Clear Interrupts
40 GOSUB 100 'Do a menu
50 GOSUB 620:MENU 'Restore F keys and exit

60 'Init Menu Prompts
70 MT$="Options Menu" 'Set up A Menu
80 M1$(1)="GC Comm": M1$(2)="System Date"
82 M1$(3)="Filer": M1$(4)=""
85 M1$(5)="": M1$(6)="": M1$(7)="": M1$(8)="Exit"
90 RETURN

100 ' Issue a menu get pick dispatch and loop
110 GOSUB 300
130 IF LF%=8 THEN RETURN
140 ON LF% GOSUB 1000,2000,3000,4000,5000,6000,7000
150 GOSUB 200:GOTO 100
180 RETURN

200 'Reset everything and Reinit Menu prompts
210 GOSUB 9000:GOSUB 60
230 RETURN

300 'Display and Read Menu
310 GOSUB 600 'Init Function Keys
320 GOSUB 340 'Display Menu
325 GOSUB 420 'Read Menu
330 RETURN

340 'Display Menu
360 CLS:? CHR$(27);"Y";CHR$(3+31);CHR$(1+31);
362 ? "Please Select:";
365 FOR M1% = 1 to 8
370 IF M1$(M1%) <> "" THEN GOSUB 390
380 NEXT M1%:RETURN
```

MENU.BA (continued)

```
390 'Display One Menu Prompt
395 L1% =INT((M1% +1) / 2):L1% = L1% + 3
398 L2% = (((M1% -1) MOD 2) * 20) +1
400 ? CHR$(27);"Y";CHR$(L1%+31);CHR$(L2%+31);
405 ? "F"+CHR$(M1%+48)+". ";M1$(M1%)
410 RETURN

420 'Read Key and check if Valid
430 GOSUB 500
440 M1% = LF%:IF M1$(M1%) = "" THEN BEEP:GOTO 430
450 RETURN

500 'Get F key input
510 LY$ =INKEY$:IF LY$ ="" THEN GOTO 510
520 LF%=ASC(LY$)
530 IF LF%<129 THEN GOTO 510
540 LF%=LF%-128:IF LF%<9 THEN RETURN
550 GOTO 510

600 'Init Function Keys
610 FOR A%=1 TO 8:KEY A%,CHR$(128+A%):NEXT A%:RETURN

620 'Restore Function Keys
630 KEY 1,"Files"+CHR$(13):KEY 2,"Load "+CHR$(34)
640 KEY 3,"Save "+CHR$(34):KEY 4, "Run"+CHR$(13)
650 KEY 5,"List "+CHR$(13): KEY 6,""
660 KEY 7,"":KEY 8,"Menu"+CHR$(13)
670 RETURN
```

MENU.BA (continued).

```
999 'Put each child program here
1000 MERGE GCM.ASM
2000 MERGE SYSDAT.ASM
3000 MERGE FILER.ASM
4000 RETURN
5000 RETURN
6000 RETURN
7000 RETURN

9000 'Disable All Interrupts & Clean
9001 'Characteristics that may be inherited
9002 'From the last child.
9010 KEY OFF:ON KEY GOSUB 0,0,0,0,0,0,0,0
9020 COM OFF:ON COM GOSUB 0
9030 TIME$ OFF: ON TIME$ GOSUB 0
9040 CLEAR:CLOSE:DEFDBL A-Z
9050 SCREEN 0,0:RETURN
```

SYSDAT.BA

```
10 'Copyright (c)1988
20 'King Computer Services, Inc.
30 'SYSDAT vers 1.0
40 'Set Date, Time and Day Using ON ERROR Logic
50 CLEAR 500:GOSUB 70:GOSUB 100
60 CLEAR:CLOSE:RETURN 'Child program must RETURN
70 MT$="Date Menu" 'Set up A Menu
80 M1$(1)="Set System Date": M1$(2)="Set System Time"
82 M1$(3)="Set System Day": M1$(4)="": M1$(5)=""
84 M1$(6)="": M1$(7)="": M1$(8)="Exit"
90 RETURN
100 GOSUB 1210:GOSUB 600:IF LF%<>8 THEN GOSUB 120
105 GOSUB 1220
110 RETURN
120 'Dispatch and then redisplay the menu
130 IF LF%=1 THEN GOSUB 190 'Set Date
140 IF LF%=2 THEN GOSUB 350 'Set Time
150 IF LF%=3 THEN GOSUB 470 'Set Day
160 GOSUB 600:IF LF%<>8 THEN GOTO 120
180 RETURN
190 GOSUB 200:GOSUB 220:RETURN 'Set Date
200 CLS:? CHR$(27);"Y";CHR$(2+31);CHR$(5+31);
205 ? "Current System Date is "+DATE$;
210 ? CHR$(27);"Y";CHR$(3+31);CHR$(1+31);
212 ? "Enter New System Date";
214 ? CHR$(27);"Y";CHR$(8+31);CHR$(35+31);:? "Exit";:RETURN
220 LE%=0:L1%=5:L2%=16:LN%=8: GOSUB 840
225 IF LF%=8 THEN RETURN
230 IF LD$="" THEN RETURN
240 GOSUB 1270
250 IF LE%=0 THEN RETURN
252 ? CHR$(27);"Y";CHR$(7+31);CHR$(1+31);
254 ? "Invalid    ";:BEEP
260 GOTO 220
350 GOSUB 360:GOSUB 380:RETURN 'Set Time
```

SYSDAT.BA (Continued)

```
360 CLS:? CHR$(27);"Y";CHR$(2+31);CHR$(5+31);
362 ? "Current System Time is "+TIME$;
370 ? CHR$(27);"Y";CHR$(3+31);CHR$(1+31);
372 ? "Enter New System Time";
374 ? CHR$(27);"Y";CHR$(8+31);CHR$(35+31);:? "Exit";:RETURN
380 LE%=0:L1%=5:L2%=16:LN%=8: GOSUB 840
385 IF LF%=8 THEN RETURN
390 IF LD$="" THEN RETURN
400 GOSUB 1430
402 IF LE%=0 THEN RETURN
410 ? CHR$(27);"Y";CHR$(7+31);CHR$(1+31);
412 ? "Invalid    ";:BEEP
420 GOTO 380
470 GOSUB 480:GOSUB 500:RETURN 'Set Day
480 CLS:? CHR$(27);"Y";CHR$(2+31);CHR$(5+31);
482 ? "Current System Day is "+DAY$;
490 ? CHR$(27);"Y";CHR$(3+31);CHR$(1+31);
492 ? "Enter New System Day";
494 ? CHR$(27);"Y";CHR$(8+31);CHR$(35+31);:? "Exit";:RETURN
500 LE%=0:L1%=5:L2%=16:LN%=3: GOSUB 840
502 IF LF%=8 THEN RETURN
510 IF LD$="" THEN RETURN
520 GOSUB 1550
530 IF LE%=0 THEN RETURN
532 ? CHR$(27);"Y";CHR$(7+31);CHR$(1+31);
534 ? "Invalid    ";:BEEP
540 GOTO 500
```

SYSDAT.BA (Continued)

```
590 'MENU.INC
600 GOSUB 1210 :LK%=1
610 H1$=DATE$: H2$=TIME$
612 H1% = INT((INSTR("MonTueWedThuFriSatSun",DAY$)) / 3)
614 H3$=DAY$
620 GOSUB 640 :GOSUB 730
630 RETURN
640 CLS:? CHR$(27);"Y";CHR$(1+31);CHR$(1+31);:? H1$;
642 ? CHR$(27);"Y";CHR$(1+31);CHR$(16+31);:? H2$;
650 ? CHR$(27);"Y";CHR$(1+31);CHR$(35+31);:? H3$;
652 ? CHR$(27);"Y";CHR$(2+31);CHR$(20-(LEN( MT$)/2)+31);
654 ? MT$;
660 ? CHR$(27);"Y";CHR$(3+31);CHR$(1+31);
662 ? "Please Select:";:FOR M1% = 1 to 8
670 IF M1$(M1%) <> "" THEN GOSUB 690 :GOSUB 710
680 NEXT M1%:RETURN
690 L1%=INT((M1%+1)/2):L1%=L1%+3:L2%=(((M1%-1)MOD2)*20)+1
700 ? CHR$(27);"Y";CHR$(L1%+31);CHR$(L2%+31);
702 ? "F"+CHR$(M1%+48)+".";:RETURN
710 L1% =INT((M1%+1)/2):L1%=L1%+3:L2%=(((M1%-1)MOD2)*20)+4
720 ? CHR$(27);"Y";CHR$(L1%+31);CHR$(L2%+31);
722 ? M1$(M1%);:RETURN
730  GOSUB 1080 :IF LF%=13 THEN BEEP:GOTO 730
740 M1% = LF%:IF M1$(M1%) = "" THEN BEEP:GOTO 730
750 RETURN
760 'SCRNIO.INC
770 LR%=0:LO%=1:RETURN
780 RETURN
790 IF LV%=1 THEN PRINT CHR$(27);"p";
800 GOSUB 830
810 IF LV%=1 THEN PRINT CHR$(27);"q";
820 LV%=0:RETURN
830 ? CHR$(27);"Y";CHR$(L1%+31);CHR$(L2%+31);
832 PRINT LD$ ;:RETURN
840 LF%=0:? CHR$(27);"Y";CHR$(L1%+31);CHR$(L2%+31);
850 IF LV$ <>"" THEN GOSUB 870
860 ? CHR$(27);"Y";CHR$(L1%+31);CHR$(L2%+31);
862 GOSUB 880:LC%=0:RETURN
870 ? CHR$(27);"Y";CHR$(L1%+31);CHR$(L2%+31);
872 ? CHR$(27);"p"; LV$;CHR$(27);"q";:LV$="":RETURN
880 LD$ ="":LS$=CHR$(13):GOSUB 1140
890 ? CHR$(27);"P";
```

SYSDAT.BA (Continued)

```
900 LY$ =INKEY$:IF LY$ = "" THEN GOTO 900
950 LP%=LEN(LD$)
952 IF LY$ >CHR$(128) OR LY$ <" " THEN GOTO 1020
970 IF LY$="," THEN BEEP :GOTO 900
980 IF LY$="\" THEN BEEP :GOTO 900
990 IF LP%<LN% THEN LD$ = LD$ + LY$:PRINT LY$;
1000 IF LP%=LN% THEN BEEP
1010 GOTO 900
1020 IF INSTR(LS$,LY$) = 0 THEN GOTO 1030
1022 LF%=ASC(LY$):? CHR$(27);"Q";
1024 IF LF%<128 THEN RETURN ELSE LF%=LF%-128:RETURN
1030 IF LY$ =CHR$(8) THEN IF LP%=0 THEN BEEP:GOTO 900
1040 IF LY$ <>CHR$(8) THEN GOTO 1050
1042 PRINT CHR$(8);" ";CHR$(8);
1044 LD$ =LEFT$( LD$ ,LP%-1):GOTO 900
1050 IF LY$ <" " OR LY$ >CHR$(127) THEN BEEP:GOTO 900
1060 IF LP%=LN% THEN BEEP:GOTO 900
1070 LD$ = LD$ + LY$ :PRINT LY$;:GOTO 900
1080 LS$="":GOSUB 1140:IF LS$="" THEN BEEP:LF%=0:RETURN
1090 ? CHR$(27);"Q";
1100 LY$ =INKEY$:IF LY$ ="" THEN GOTO 1100
1110 LF%=INSTR(LS$,LY$):IF LF% = 0 THEN BEEP:GOTO 1100
1120 LF%=ASC(LY$):IF LF%>128 THEN LF%=LF%-128
1130 RETURN
1140 IF LK%=0 THEN GOTO 1150
1142 LS$=LS$ +CHR$(129)+CHR$(130)+CHR$(131)+CHR$(132)
1144 LS$=LS$+ CHR$(133)+CHR$(134)+CHR$(135)+CHR$(136)
1150 IF LU%=1 THEN LS$ = LS$ +CHR$(30)+CHR$(31)
1160 IF LW%=1 THEN LS$ = LS$ +CHR$(28)+CHR$(29)
1170 IF LM%=1 THEN LS$=LS$+"0123456789"
1180 RETURN
1190  LY$ =INKEY$:IF LY$ ="" THEN GOTO 1190
1200 RETURN
```

SYSDAT.BA (Continued)

```
1210 FOR A%=1 TO 8
1212 KEY A%,CHR$(128+A%)
1214 NEXT A%
1218 RETURN
1220 KEY 1,"Files"+CHR$(13):KEY 2,"Load "+CHR$(34)
1222 KEY 3,"Save "+CHR$(34):KEY 4, "Run"+CHR$(13)
1230 KEY 5,"List "+CHR$(13): KEY 6,"":KEY 7,""
1232 KEY 8,"Menu"+CHR$(13)
1240 RETURN
1260 '
1270 ON ERROR GOTO 1330
1280 DATE$=LD$
1290 ON ERROR GOTO 1310
1300 RETURN
1310 ? "Error ";ERR; " at ";ERL
1320 END
1330 LE%=1
1340 RESUME NEXT
1400 'The assignment statements that require trapping
1410 'are placed at the end of the code so that error
1420 'trapping logic is generated at the end of the prog
1430 ON ERROR GOTO 1330
1440 TIME$=LD$
1450 ON ERROR GOTO 1310
1460 RETURN
1550 ON ERROR GOTO 1330
1560 DAY$=LD$
1570 ON ERROR GOTO 1310
1580 RETURN
```

Compiling for the Model 200

APPLIES TO VERSION 5.0 ONLY

If you have acquired the Model 200 version of the compiler, your distribution diskette will contain 3 extra files, ROMHEAD.200, SYSAD.200 and RCO200.ASM.

These two files along with some compiler options make it possible to compile programs for the Tandy Model 200.

The RBASIC compiler starting with version 3.00 supports a /2 switch that informs the compiler that the program is to be compiled for the Model 200. This causes the compiler to include the SYSAD.200 file instead of SYSAD.100 at compile time. If the program is being compiled for ROM development, then the ROMHEAD.200 file will be included instead of ROMHEAD.100.

The A100 assembler starting with version 3.00 includes a /2 switch that informs the assembler to use maxram values for the 200 instead of the 100 when it does the final calculations of highest possible org address and checks whether a program has gone in to high memory.

Please note that the compiler does NOT translate CALLS within the program. If a program for the Model 102 contained a line to call the routine to connect the phone for the Model 102, neither the compiler nor the assembler will attempt to translate that value to the Model 200 equivalent. Those translations are up to the programmer.

```
200 CALL 21200 'Connect the phone
```

If a program PROGA.BA does not contain machine specific calls, it can be compiled and assembled for the Model 100/102 with

```
RBASIC PROGA
A100 PROGA
```

The same program will compile and assemble for the Model 200 by using the /2 switch.

```
RBASIC PROGA /2
A100 PROGA /2
```

Receiving on the Model 200

RCO200.ASM is a modified version of RCO.ASM designed to work with the Model 200. It is used just like RCO.ASM for the Model 100/102.

In order to get a .HEX file onto the 200 as a machine language program, a program must exist on the Model 200 that can receive and load this type of file. This .HEX loader should be as efficient as possible. A HEX loader could be written in BASIC, but BASIC runs very slowly when you are trying to load large files. The answer is to create a HEX loader that is written in machine language.

COLOADER.EXE handles this nicely. COLOADER is a program that runs on the PC. It will write a BASIC program that can be downloaded to the Model 200. The BASIC program is not the HEX loader, but when it is run on the Model 200 it will create a machine language program that is a HEX loader.

This sounds complicated, but it simple to use.

Using COLOADER.EXE

It is easiest to describe COLOADER.EXE using a real example. Assume you have created a BASIC program PROGA.BA that must be ORGed at 57000 to fit in to Model 200 memory.

The program would be compiled and assembled using that address.

```
RBASIC PROGA /o57000 /2

A100 PROGA /2
```

The output from these two steps will be a file named PROGA.HEX.

Now we're ready to use COLOADER.

1. Ensure that the files COLOADER.EXE, A100.EXE and RCO200.ASM are available in the current directory.

2. Calculate a second ORG address that is 500 bytes below the address of your program.

   57000 - 500 = 56500

3. Using this address, start COLOADER.EXE with the following command.

   COLOADER RCO200 /o56500

   This will create a BASIC program named RCO200.BA.

4. Load and run RCO200.BA on the Model 200. On the Model 200 type

   RUN "COM:88N1ENN" <ENTER>

   On the PC type

   SND RCO200.BA <ENTER>

5. While RCO200.BA is running it will display the message "Creating RCO200.CO Program". This should only take a few seconds, and the 'Ok' Prompt will appear.

6. The BASIC program that was just loaded and run has now done its job and you may delete it by typing "NEW" and pressing ENTER.

7. Press F8 to return to the Model 200. The menu will now contain the program RCO200.CO. This program is the HEX loader and can be used to load your program as described in the following steps.

8. Position the cursor over RCO200.CO and press enter. The Model 200 Screen will clear, and the loader is now waiting for you to send your program file.

9. On the PC type

   SND PROGA.HEX <ENTER>

10. While your program is loading, RCO200.CO will display dots ('.') on the screen indicating that HEX records are being received and poked into the Model 200 Memory.

11. When the load is complete, you will return to the Model 200 System Menu.

12. To start running your program, enter BASIC and type

    CALL 57000 <ENTER>

    As long as your program's original ORG address does not change, you may edit, recompile and reload using RCO200.CO as many times as needed. If your program must be ORGed at a lower address because of an increase in size, then go back to Step 1 and create a new version of RCO200.BA and download and run it.

Version 5 Enhancements

There have been several improvements in version 5 beyond those discussed in the preceding two chapters. These are advanced programming concepts and are mainly aimed at tightening the code generated by RBASIC when special circumstances exist.

New Switches

Resume Jump Switch - /J

ON ERROR handling allows for three types of RESUME statements.

RESUME causes the program to re-execute the the line that caused the problem.

RESUME NEXT causes the program to to resume execution at the next statement after the statement that caused the error.

RESUME (line) as in RESUME 1540, causes the program to resume control at a specific line number.

If an error occurs during execution, the running program has to have been prepared for the error handling by saving the address of the current statement in the event of a RESUME command, and the address of the next statement in the event of a RESUME NEXT.

As you can imagine, the process of saving the current program line and the next program line before each line is executed causes the size of the code to be increased.

Some programs only use, or only require a RESUME (LINE) directive to recover from an error. If this is the case, it is not necessary to save current and next lines. If you use this type of error recovery, then use the /J switch on the command line instead of the standard /E switch.

The /E switch also turns on line numbering, but the /J switch does not, so if you do need to retain line numbering then use both switches /J and /L.

The combination of /J and /L is still smaller and executes slightly faster than the /E switch.

The /J switch must NOT be used if you use RESUME or RESUME NEXT for error recovery. Use the /E switch for these. The compiler does not check this, and will assume a RESUME (LINE) type recovery if you use the /J switch.

Non-KEY Interrupts /X.

Under version 4 of RBASIC ON COM, ON TIME$ and ON KEY interrupts were all treated in the same way. If your program contained any one of these interrupts, or was compiled with the /I switch, then all three interrupts were enabled. This caused an unexpected side effect. The normal compiled code for RBASIC disables the Break Key checking. This break key checking could be turned back on by using the /B switch, but many developers liked the idea of disabling the key.

The problem was that ON KEY logic would turn the break check back on. The result was that if you wanted to disable the break key, but used an ON TIME$ or ON COM interrupt, the break key was enabled to as part of the ON KEY check.

Version 5 recognizes the difference between ON KEY and the other two types of interrupts. ON KEY interrupt checking is only turned on if the /I switch is used, or an ON KEY is countered in the program. The /X switch can be used as an alternative to the /I switch to enable ON COM and ON TIME$ checking only.

If you want to allow the compiler to turn on checking, then do not use either switch. If you are using only ON COM and/or ON TIME$ then use the /X switch. If you are using ON KEY then use the /I switch.

Further information is available in the chapter on Interrupt processing.

Compiler Control Directives.

Version 5 has taken advantage of another unused BASIC key word, CONT, to implement compiler control statements. This command is used to change the way the compiler behaves, and does not act as a BASIC command. You can consider the CONT keyword to be a mnemonic for CONTROL.

Compiler directives are useful for debugging and error trapping.

The compiler control directives are:

```
CONT LINE ON      Start Line numbering
CONT LINE END     Stop  Line Numbering

CONT TRACE ON     Start including Trace logic
CONT TRACE OFF    Stop  including trace logic.

CONT SLOW ON      Start including slow trace logic
CONT SLOW OFF     Stop  including trace logic.
```

The uses of line numbering, trace and slow trace logic are described in the chapter on compiler switches. The advantage of compiler directives is that you can control the behavior over a smaller range of lines than the whole program.

For example if you are trying to debug one section of code, use the TRACE ON and END around the code to to provide a limited trace.

```
100 'ORIGINAL ROUTINE
110 'DID SOMETHING HERE
120 'DOES SOMETHING ELSE AND FINALLY
130 RETURN
```

becomes

```
100 CONT TRACE ON: 'ORIGINAL ROUTINE
110 'DID SOMETHING HERE
120 'DOES SOMETHING ELSE AND FINALLY
130 CONT TRACE END: RETURN
```

The LINE directive is very useful. In most programs it is not necessary to retain line numbering for the entire program, rather it is usually only needed for a few critical routines.

Use CONT LINE ON and END around these routines to ensure that line numbers are available in the vent of errors.

The compiler CONTROL directives override compile time switches.

For example:

```
10 CONT LINE END
```

If a program started with line 10 above and were compiled with

```
RBASIC /l prog
```

Line numbering would be turned off by the first line of the program.

This can be most effectively used with the /Error switch. If ON ERROR logic is needed, but line numbering is not, or line numbering is only needed in certain critical areas then

```
10 CONT LINE OFF
20 ' PROGRAM here
.......
.......
500 'CRITICAL ROUTINE STARTS HERE
510 CONT LINE ON
.........
.........
590 RETURN
595 CONT LINE END
```

ROMS and HEX Files

Selecting and Using EPROMs.

If you have no experience with ROMS/EPROMS there are several good books on the general problem of EPROM burning. One such is "Experiments With EPROMS" by Dave Prochnow published by Tab Books, Blue Ridge Summit, PA 17294.

The EPROM (Erasable Programable Read Only Memory) that should be used for the Model 100/102 is the 27C256. The C denotes low power CMOS. This is a 32K ROM. There are smaller ROMs such as the 16K, 27C128 and the 8K 27C64 but the price difference between them is so small that it has never seemed worth the effort to stock up of three different sizes.

The other factor that comes up in choosing EPROMS is memory speed. The Model 100/102 is very slow and apparently will not tax even the slowest EPROM. I have used 250 ns (nano-second) EPROMS and I have been told by an engineer that units as slow as 300 or 350 ns would still work. The 250 ns speed is fairly common.

It is ok to use faster EPROMS if that is all you can get but it will not make youir program run faster and once access rates go below 150 nanoseconds, prices start to rise. The speed of an EPROM is frequently incorporated in the part number by including a dash followed by the speed in tens of nano-seconds thus

27C256-15	150 nanoseconds
27C256-25	250 nanoseconds
27C256-5	50 nanoseconds
27C256-10	100 nanoseconds

Burn voltage is usually 12.5 volts. Ask your supplier to be certain.

Once the EPROM is burned it cannot fit directly into a ROM socket because the pin out is different between the socket and the chip.

Radio Shack sells an adapter that will change the pin outs. The EPROM chip must be soldered to the adapter. The adapter can be ordered through your local Radio Shack and is Part Number AXX-7113 and is called the Tandy 100/200 EPROM Adapter.

It costs just over $10.00 and comes complete with soldering and installation instructions.

Intel HEX format.

The .HEX files created as output by the A100 assembler are a modified Intellec 8/MDS Code 83 format.

Strictly speaking this format is supposed to use a final record formatted as follows:

```
:00000001FF
```

The assembler generates a CPM style all zeroes final record of

```
:0000000000
```

This format has been used with several different ROM burners and none of them have failed to recognize the last record.

King Computer Services, Inc.
1016 North New Hampshire
Los Angeles, CA 90029
(213) 661-2063

September 27, 1988

ROM CARRIAGE ASSEMBLY

The ROM may be soldered directly to the circuit board.

Alternatively, a removable circuit board may be created by soldering special pins to the circuit board, so that the ROM can be easily inserted and removed.

The ROM or pins are inserted into the side of the circuit board marked "component side" and are soldered on the reverse.

Pin 1 is marked on the circuit board on the edge of the card away from the two protusions used as insertion guides. These reverse notches are used for inserting into the molex socket which has guide channels.

The circuit board requires a spacer of approximately 1/10"on the circuit side.

The grey plastic strips are used to create spacers by cutting off two lengths about 1/4" shorter than the carriages and gluing them to the circuit side of the board so that the space the carriage about 1/10" from the molex socket on the M100. They should be glued with super glue or crazy glue.

Place Eprom legs through holes
in Circuit board as shown.
Solder legs to boards
Super Glue the spacer board to
the bottom of the unit,